

Documentación de lenguaje C++

Aprenda a usar C++ y la biblioteca estándar de C++.

Información sobre C++ en Visual Studio

DESCARGAR

[Descargar Visual Studio para Windows ↗](#)

[Instalar compatibilidad con C/C++ en Visual Studio](#)

[Descargar solo las herramientas de compilación de línea de comandos ↗](#)

INTRODUCCIÓN

[Hello World en Visual Studio con C++](#)

[Creación de una calculadora de consola en C++](#)

VIDEO

[Información sobre C++ : lenguaje y biblioteca de uso general](#)

CURSOS

[Aquí está otra vez C++: C++ moderno](#)

[Ejemplos y archivo de ejemplo](#)

Novedades de C++ en Visual Studio

NOVEDADES

[Novedades de C++ en Visual Studio](#)

[Mejoras de conformidad de C++](#)

INFORMACIÓN GENERAL

[Introducción al desarrollo de C++ en Visual Studio](#)

[Plataformas de destino compatibles](#)

[Ayuda y recursos de la comunidad](#)

[Notificar un problema o hacer una sugerencia](#)

Usar el compilador y las herramientas

REFERENCIA

[Referencia de compilación de C/C++](#)

[Proyectos y sistemas de compilación](#)

[Referencia del compilador](#)

[Referencia del enlazador](#)

[Herramientas de compilación adicionales](#)

[Errores y advertencias](#)

Lenguaje C++

REFERENCIA

[Referencia del lenguaje C++](#)

[Palabras clave de C++](#)

[Operadores de C++](#)

[Referencia del preprocesador de C/C++](#)

Biblioteca estándar de C++ (STL)

REFERENCIA

[Información general sobre la biblioteca estándar de C++](#)

[Referencia de la biblioteca estándar de C++ por encabezado](#)

[Contenedores de la biblioteca estándar de C++](#)

[Iteradores](#)

[Algoritmos](#)

[Asignadores](#)

Objetos de función

programación con iostream

Expresiones regulares

Exploración del sistema de archivos

Referencia del lenguaje C++

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

En esta referencia se explica el lenguaje de programación C++ tal como se implementa en el compilador de Microsoft C++. La organización se basa en el *manual de referencia de C++ anotado* de Margaret Ellis y Bjarne Stroustrup, y en los estándares internacionales ANSI/ISO C++ (ISO/IEC FDIS 14882). Se incluyen las implementaciones específicas de Microsoft de las características del lenguaje C++.

Para obtener información general sobre las prácticas de programación modernas de C++, consulte [Bienvenido a C++](#).

Consulte las tablas siguientes para encontrar rápidamente una palabra clave o un operador:

- [Palabras clave de C++](#)
- [Operadores de C++](#)

En esta sección

[Convenciones léxicas](#)

Elementos léxicos fundamentales de un programa de C++: tokens, comentarios, operadores, palabras clave, signos de puntuación, literales. También, traducción de archivos, prioridad o asociatividad de los operadores.

[Conceptos básicos](#)

Ámbito, vinculación, inicio y finalización del programa, clases de almacenamiento y tipos.

Tipos integrados Los tipos fundamentales integrados en el compilador de C++ y sus intervalos de valores.

[Conversiones estándar](#)

Conversiones de tipos entre tipos integrados También, conversiones aritméticas y conversiones entre tipos de puntero, referencia y puntero a miembro.

Declaraciones y definiciones Declarar y definir variables, tipos y funciones.

[Operadores, prioridad y asociatividad](#)

Operadores de C++.

Expresiones

Tipos de expresiones, semántica de expresiones, temas de referencia sobre operadores, conversión y operadores de conversión, información de tipos en tiempo de ejecución.

Expresiones lambda

Una técnica de programación que define implícitamente una clase de objeto de función y crea un objeto de función de ese tipo de clase.

Instrucciones

Instrucciones de expresión, null, compuestas, de selección, de iteración, de salto y de declaración.

Clases y estructuras

Introducción a las clases, estructuras y uniones. También, funciones miembro, miembros de datos, campos de bits, puntero `this`, clases anidadas.

Uniones

Tipos definidos por el usuario en los que todos los miembros comparten la misma ubicación de memoria.

Clases derivadas

Herencia sencilla y múltiple, funciones `virtual`, clases base múltiples, clases `abstractas`, reglas de ámbito. También las palabras clave `__super` y `__interface`.

Control de acceso a miembros

Controlar el acceso a los miembros de clase: palabras clave `public`, `private` y `protected`. Funciones y clases friend.

Sobrecarga

Operadores sobrecargados, reglas para la sobrecarga de operadores.

Control de excepciones

Control de excepciones de C++, control estructurado de excepciones (SEH), palabras clave usadas para escribir instrucciones de control de excepciones.

Aserción y mensajes proporcionados por el usuario

La directiva `#error`, la palabra clave `static_assert`, la macro `assert`.

Templates (Plantillas [C++])

Especificaciones de plantilla, plantillas de función, plantillas de clase, palabra clave `typename`, plantillas y macros, plantillas y punteros inteligentes.

Control de eventos

Declaración de eventos y controladores de eventos.

[Modificadores específicos de Microsoft](#)

Modificadores específicos de Microsoft C++. Dirección de memoria, convenciones de llamada, funciones `naked`, atributos extendidos storage-class (`__declspec`), `__w64`.

[Ensamblador insertado](#)

Uso del lenguaje de ensamblado y C++ en bloques `__asm`.

[Compatibilidad con COM del compilador](#)

Una referencia a las clases específicas de Microsoft y funciones globales utilizadas para admitir tipos COM.

[Extensiones para Microsoft](#)

Extensiones de Microsoft a C++.

[Comportamiento no estándar](#)

Información sobre el comportamiento no estándar del compilador de Microsoft C++.

[Aquí está otra vez C++](#)

Información general sobre las prácticas de programación modernas de C++ para escribir programas seguros, correctos y eficientes.

Secciones relacionadas

[Extensiones de componentes para plataformas de tiempo de ejecución](#)

Material de referencia sobre el uso del compilador de Microsoft C++ para tener como destino .NET.

[Referencia de compilación de C/C++](#)

Opciones del compilador, opciones del vinculador y otras herramientas de compilación.

[Referencia del preprocesador de C/C++](#)

Material de referencia sobre instrucciones pragma, directivas de preprocesador, macros predefinidas y el preprocesador.

[Bibliotecas de Visual C++](#)

Una lista de vínculos a las páginas de inicio de referencia para las diferentes bibliotecas de Microsoft C++.

Vea también

[Referencia del lenguaje C](#)

Aquí está otra vez C++: C++ moderno

Artículo • 03/03/2023 • Tiempo de lectura: 10 minutos

Desde su creación, C++ se ha convertido en uno de los lenguajes de programación más utilizados en el mundo. Los programas bien escritos de C++ son rápidos y eficaces. Este lenguaje es más flexible que otros: puede funcionar en los niveles más altos de abstracción o bajar al nivel del silicio. C++ proporciona bibliotecas estándar altamente optimizadas. Asimismo, permite el acceso a características de hardware de bajo nivel para maximizar la velocidad y minimizar los requisitos de memoria. C++ puede crear casi cualquier tipo de programa: juegos, controladores de dispositivos, HPC, nube, escritorio, incrustado, aplicaciones móviles, etc. Incluso hay bibliotecas y compiladores de otros lenguajes de programación escritos en C++.

Uno de los requisitos originales para C++ era la compatibilidad con el lenguaje C. Como resultado, C++ siempre ha permitido la programación de estilo C, con punteros básicos, matrices, cadenas de caracteres terminadas en NULL y otras características. Dichas características ofrecen un gran rendimiento, pero también pueden generar errores y complejidades. La evolución de C++ tiene características destacadas que reducen en gran medida la necesidad de utilizar expresiones de estilo C. Las antiguas características de programación C todavía están ahí cuando las necesita. Sin embargo, en código C++ moderno debería necesitarlas cada vez menos. El código C++ moderno es más sencillo, más seguro y más elegante, y tan rápido como siempre.

En las secciones siguientes se proporciona información general sobre las características principales de C++ moderno. A menos que se indique lo contrario, las características que se enumeran aquí están disponibles en C++11 y versiones posteriores. En el compilador de Microsoft C++, puede establecer la opción de compilador `/std` para especificar qué versión del estándar se usará para el proyecto.

Recursos y punteros inteligentes

Una de las principales clases de errores en la programación de estilo C es la *fuga de memoria*. A menudo, las fugas se deben a un error al realizar llamadas a `delete` para memoria que se ha asignado con `new`. En el código C++ moderno destaca el principio de que *la adquisición de recursos es la inicialización* (RAII). La idea es sencilla. Los recursos (memoria de montón, identificadores de archivos, sockets, etc.) deben ser *propiedad* de un objeto. Ese objeto crea o recibe el recurso recién asignado en su constructor y lo elimina en su destructor. El principio de RAII garantiza que todos los recursos se devuelvan correctamente al sistema operativo cuando el objeto propietario salga del ámbito.

Para admitir la adopción sencilla de los principios de RAII, la biblioteca estándar de C++ proporciona tres tipos de puntero inteligente: `std::unique_ptr`, `std::shared_ptr` y `std::weak_ptr`. Un puntero inteligente controla la asignación y la eliminación de la memoria de la que es propietario. En el ejemplo siguiente se muestra una clase con un miembro de matriz que se asigna en el montón en la llamada a `make_unique()`. La clase encapsula las llamadas a `new` y `unique_ptr delete`. Cuando un objeto `widget` sale del ámbito, se invoca el destructor `unique_ptr`, el cual liberará la memoria asignada para la matriz.

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
```

Siempre que sea posible, use un puntero inteligente al administrar memoria en montón. Si debe usar los operadores `new` y `delete` explícitamente, siga el principio de RAII. Para obtener más información, vea [Duración de objetos y administración de recursos \(RAII\)](#).

std::string y std::string_view

Las cadenas de estilo C son otra de las principales fuentes de errores. Mediante el uso de `std::string` y `std::wstring`, puede eliminar prácticamente todos los errores asociados a las cadenas de estilo C. También podrá aprovechar las ventajas de las funciones miembro para buscar, anexar y anteponer elementos, entre otras tareas. Ambos están muy optimizados para la velocidad. Al pasar una cadena a una función que únicamente requiere acceso de solo lectura, en C++17 puede usar `std::string_view` para obtener una ventaja de rendimiento incluso mayor.

`std::vector` y otros contenedores de la biblioteca estándar

Todos los contenedores de la biblioteca estándar siguen el principio de RAII y proporcionan iteradores para el recorrido seguro de los elementos. Además, están muy optimizados para el rendimiento y se han sometido a pruebas exhaustivas para comprobar si son correctos. Mediante el uso de estos contenedores, se elimina la posibilidad de que haya errores o ineficiencias que podrían transferirse a estructuras de datos personalizadas. En lugar de matrices sin formato, use `vector` como un contenedor secuencial en C++.

C++

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

Use `map` (no `unordered_map`) como contenedor asociativo predeterminado. Use `set`, `multimap` y `multiset` para los casos degenerados y múltiples.

C++

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

Cuando la optimización del rendimiento es necesaria, considere utilizar:

- Tipo `array`, cuando la incrustación es importante, por ejemplo, como miembro de clase.
- Contenedores asociativos desordenados, como `unordered_map`. Estos tienen una menor sobrecarga por elemento y una búsqueda de tiempo constante, pero pueden ser más difíciles de usar de forma correcta y eficaz.
- Elementos `vector` ordenados. Para más información, vea [Algoritmos](#).

No use matrices de estilo C. En el caso de las API más antiguas que necesiten acceso directo a los datos, use mecanismos de acceso como `f(vec.data(), vec.size());` en su lugar. Para obtener más información sobre los contenedores, vea [Contenedores de la biblioteca estándar de C++](#).

Algoritmos de biblioteca estándar

Antes de suponer que necesita escribir un algoritmo personalizado para el programa, revise primero los **algoritmos** de la biblioteca estándar de C++. La biblioteca estándar contiene una serie de algoritmos en constante crecimiento para muchas operaciones comunes, como la búsqueda, la ordenación, el filtrado o la aleatorización. La biblioteca matemática es muy amplia. A partir de C++17 se proporcionan versiones paralelas de muchos algoritmos.

Aquí se describen algunos ejemplos importantes:

- `for_each`, el algoritmo de recorrido predeterminado (junto con los bucles `for` basados en rangos).
- `transform`, para la modificación descontextualizada de los elementos de contenedor.
- `find_if`, el algoritmo de búsqueda predeterminado.
- `sort`, `lower_bound`, y los demás algoritmos de ordenación y búsqueda predeterminados.

Para escribir un comparador, utilice un elemento `<` estricto y *lambdas con nombre* cuando pueda.

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 );
```

auto en lugar de nombres de tipos explícitos

C++11 incluyó por primera vez la palabra clave `auto` para su uso en declaraciones de variables, funciones y plantillas. `auto` indica al compilador que deduzca el tipo del objeto para que no tenga que escribirlo explícitamente. `auto` es especialmente útil cuando el tipo deducido es una plantilla anidada:

C++

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

Bucles `for` basados en rangos

La iteración de estilo C sobre matrices y contenedores es propensa a la indexación de errores, además de tediosa de escribir. Para eliminar estos errores y hacer que el código sea más legible, use bucles `for` basados en rangos con contenedores de la biblioteca estándar y matrices sin formato. Para obtener más información, vea [Instrucción for basada en intervalo \(C++\)](#).

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;
    }
}
```

Expresiones `constexpr` en lugar de macros

Las macros en C y C++ son tokens procesados por el preprocesador antes de la compilación. Todas las instancias de un token de macro se reemplazan por su valor o expresión definidos antes de la compilación del archivo. Las macros se utilizan normalmente en la programación de estilo C para definir valores constantes en tiempo de compilación. Sin embargo, las macros son propensas a errores y difíciles de depurar. En C++ moderno, debería dar preferencia a las variables `constexpr` para las constantes en tiempo de compilación:

C++

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

Inicialización uniforme

En C++ moderno, puede usar la inicialización de llaves para cualquier tipo. Esta forma de inicialización es especialmente útil al inicializar matrices, vectores u otros contenedores. En el ejemplo siguiente, `v2` se inicializa con tres instancias de `S`. `v3` se inicializa con tres instancias de `S` que, a su vez, se inicializan mediante llaves. El compilador infiere el tipo de cada elemento según el tipo declarado de `v3`.

```
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} };

}
```

Para obtener más información, vea [Inicialización de llaves](#).

Semántica de transferencia de recursos

C++ moderno proporciona *semántica de transferencia de recursos*, lo que permite eliminar copias de memoria innecesarias. En versiones anteriores del lenguaje, las copias eran inevitables en determinadas situaciones. Una operación *move* transfiere la propiedad de un recurso de un objeto al siguiente sin hacer una copia. Algunas clases

son propietarias de recursos como memoria de montón, identificadores de archivo y otros elementos. Cuando se implementa una clase propietaria de recursos, se puede definir un *constructor de movimiento* y un *operador de asignación de movimiento* para ella. El compilador elige estos miembros especiales durante la resolución de sobrecargas en situaciones en las que no se necesita una copia. Los tipos de contenedor de la biblioteca estándar invocan al constructor de movimiento en objetos si se define uno. Para obtener más información, vea [Constructores de movimiento y operadores de asignación de movimiento \(C++\)](#).

Expresiones lambda

En la programación de estilo C, se puede pasar una función a otra mediante un *puntero de función*. El mantenimiento y la comprensión de los punteros de función no es sencilla. La función a la que hacen referencia puede definirse en cualquier parte del código fuente, lejos del punto en el que se invoca. Además, no cuentan con seguridad de tipos. C++ moderno proporciona *objetos de función*, que son clases que invalidan el operador [operator\(\)](#), lo que permite que se les llame como una función. La forma más práctica de crear objetos de función es con [expresiones lambda](#) insertadas. En el ejemplo siguiente se muestra cómo usar una expresión lambda para pasar un objeto de función que la función `find_if` invocará en los elementos del vector:

```
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; });
```

La expresión lambda `[=](int i) { return i > x && i < y; }` se puede leer como "función que toma un único argumento de tipo `int` y devuelve un valor booleano que indica si el argumento es mayor que `x` y menor que `y`". Observe que las variables `x` y `y` del contexto circundante se pueden usar en la expresión lambda. El símbolo `[=]` especifica que el valor *captura* esas variables; es decir, la expresión lambda tiene sus propias copias de dichos valores.

Excepciones

C++ moderno destaca las excepciones en lugar de los códigos de error como la mejor manera de notificar y controlar las condiciones de error. Para obtener más información,

vea Procedimientos recomendados de C++ moderno para excepciones y control de errores.

std::atomic

Use el struct [std::atomic](#) y los tipos relacionados de la biblioteca estándar de C++ para los mecanismos de comunicación entre subprocessos.

std::variant (C++17)

Las uniones se suelen usar en la programación de estilo C para conservar memoria, ya que permiten que los miembros de tipos diferentes ocupen la misma ubicación de memoria. Sin embargo, las uniones no cuentan con seguridad de tipos y son propensas a errores de programación. C++17 incluye por primera vez la clase [std::variant](#) como una alternativa más sólida y segura a las uniones. La función [std::visit](#) se puede usar para acceder a los miembros de un tipo `variant` con seguridad de tipos.

Vea también

[Referencia del lenguaje C++](#)

[Expresiones lambda](#)

[Biblioteca estándar de C++](#)

[Conformidad del lenguaje Microsoft C/C++](#)

Convenciones léxicas

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

En esta sección se presentan los elementos fundamentales de un programa de C++. Estos elementos, denominados "elementos léxicos" o "tokens", se usan para construir instrucciones, definiciones, declaraciones, etc., que, a su vez, se usan para construir programas completos. Los elementos léxicos siguientes se tratan en esta sección:

- [Juegos de tokens y caracteres](#)
- [Comentarios](#)
- [Identificadores](#)
- [Palabras clave](#)
- [Signos de puntuación](#)
- [Literales numéricos, booleanos y de puntero](#)
- [Literales de cadena y carácter](#)
- [Literales definidos por el usuario](#)

Para obtener más información sobre cómo se analizan los archivos de código fuente de C++, consulte [Fases de traducción](#).

Vea también

[Referencia del lenguaje C++](#)

[Unidades de traducción y vinculación](#)

Juegos de tokens y caracteres

Artículo • 03/03/2023 • Tiempo de lectura: 4 minutos

El texto de un programa de C++ consta de tokens y *espacios en blanco*. Un token es el elemento mínimo de un programa de C++ que es significativo para el compilador. El analizador de C++ reconoce estos tipos de tokens:

- Palabras clave
- Identificadores
- Literales numéricos, booleanos y de puntero
- Literales de cadena y carácter
- Literales definidos por el usuario
- Operadores
- Signos de puntuación

Los tokens suelen estar separados por *espacios en blanco*, que pueden ser uno o varios:

- Espacios en blanco
- Tabulaciones horizontales o verticales
- Nuevas líneas
- Fuentes de formularios
- Comentarios

Juego básico de caracteres de código fuente

El estándar de C++ especifica un *juego de caracteres de origen básico* que se puede usar en los archivos de código fuente. Para representar caracteres ajenos a este conjunto, los caracteres adicionales se pueden especificar mediante el uso de un *nombre de carácter universal*. La implementación de MSVC permite caracteres adicionales. El *juego de caracteres de origen básico* consta de 96 caracteres que pueden usarse en archivos de código fuente. Este conjunto incluye el carácter de espacio, tabulación horizontal, tabulación vertical, avance de página y caracteres de control de nueva línea, además del siguiente conjunto de caracteres gráficos:

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

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

Específicos de Microsoft

MSVC incluye el carácter `$` como miembro del juego de caracteres de origen básico. MSVC también permite usar un juego de caracteres en archivos de código fuente, con base en la codificación del archivo. De forma predeterminada, Visual Studio almacena archivos de código fuente mediante la página de códigos predeterminada. Al guardar los archivos de código fuente mediante una página de códigos específicos de la configuración regional o una página de códigos Unicode, MSVC le permite usar cualquiera de los caracteres de dicha página de códigos en el código fuente, excepto los códigos de control no permitidos explícitamente en el juego de caracteres de origen básico. Por ejemplo, se pueden colocar caracteres de japonés en comentarios, identificadores o literales de cadena, si se guarda el archivo a través de una página de códigos de japonés. MSVC no permite las secuencias de caracteres que no pueden trasladarse a caracteres multibyte o puntos de código Unicode válidos. Según las opciones del compilador, puede que no todos los caracteres permitidos se muestren en los identificadores. Para obtener más información, vea [Identificadores](#).

FIN de Específicos de Microsoft

nombres de carácter universal

Dado que los programas de C++ pueden usar muchos más caracteres que los especificados en el juego básico de caracteres de código fuente, es posible especificar estos caracteres de una manera portátil mediante *nombres de carácter universal*. Un nombre de carácter universal consta de una secuencia de caracteres que representan un punto de código Unicode. Estos tienen dos formatos. Use `\UNNNNNNNN` para representar un punto de código Unicode con el formato U+NNNNNNNN, donde NNNNNNNN es el número de punto de código hexadecimal de ocho dígitos. Use `\uNNNN` de cuatro dígitos para representar un punto de código Unicode con el formato U+0000NNNN.

Los nombres de carácter universal pueden usarse tanto en identificadores como en literales de cadena y carácter. Un nombre de carácter universal no puede usarse para representar un punto de código suplente en el rango de 0xD800 a 0xDFFF. En su lugar, se debe usar el punto de código deseado. El compilador genera automáticamente los suplentes necesarios. Se aplican restricciones adicionales a los nombres de carácter universal que se pueden usar en los identificadores. Para obtener más información, vea [Identifiers](#) y [String and Character Literals](#).

Específicos de Microsoft

El compilador de Microsoft C++ trata indistintamente un carácter con el formato de nombre de carácter universal y con formato de literal. Por ejemplo, se puede declarar un

identificador con formato de nombre de carácter universal y usarlo en el formato de literal:

C++

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

El formato de caracteres extendidos en el Portapapeles de Windows es específico de la configuración regional de la aplicación. Cortar y pegar estos caracteres en el código desde otra aplicación podría introducir codificaciones de caracteres inesperadas. Esto puede provocar errores de análisis sin causa visible en el código. Se recomienda establecer la codificación del archivo de código fuente en una página de códigos Unicode antes de pegar los caracteres extendidos. También se recomienda usar un IME o la aplicación Mapa de caracteres para generar caracteres extendidos.

FIN de Específicos de Microsoft

Juegos de caracteres de ejecución

Los *juegos de caracteres de ejecución* representan los caracteres y las cadenas que pueden aparecer en un programa compilado. Estos juegos de caracteres constan de todos los caracteres permitidos en un archivo de código fuente, así como de los caracteres de control que representan alertas, retrocesos, retorno de carro y el carácter nulo. El juego de caracteres de ejecución tiene una representación específica de la configuración regional.

Comentarios (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Un comentario es texto que el compilador omite pero que es útil para los programadores. Los comentarios se usan normalmente para anotar código para su referencia futura. El compilador los trata como si fueran espacios en blanco. Puede usar comentarios en las pruebas para desactivar algunas líneas de código; sin embargo, para esto es mejor utilizar directivas de preprocesador `#if/#endif` porque se puede incluir entre ellas código que contiene comentarios, pero no se pueden anidar comentarios.

Los comentarios de C++ se escriben de una de las maneras siguientes:

- Los caracteres `/*` (barra diagonal, asterisco), seguidos de cualquier secuencia de caracteres (incluidas nuevas líneas), seguidos de los caracteres `*/`. Esta sintaxis es la misma que para ANSI C.
- Los caracteres `//` (dos barras diagonales), seguidos de cualquier secuencia de caracteres. Una nueva línea que no va precedida inmediatamente de una barra diagonal inversa finaliza esta forma de comentario. Por tanto, normalmente se denomina "comentario de una sola línea".

Los caracteres de comentario (`/*`, `*/` y `//`) no tienen ningún significado especial dentro de una constante de caracteres, un literal de cadena o un comentario. Por tanto, los comentarios que usan la primera sintaxis no se pueden anidar.

Consulte también

[Convenciones léxicas](#)

Identificadores (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

Un identificador es una secuencia de caracteres que se usa para denotar:

- El nombre de un objeto o variable
- Un nombre de clase, estructura o unión
- Un nombre de tipo enumerado
- El miembro de una clase, estructura, unión o enumeración
- Una función o una función miembro de clase
- Un nombre de typedef
- Un nombre de etiqueta
- Un nombre de macro
- Un parámetro de macro

Los siguientes caracteres son válidos como cualquier carácter de un identificador:

```
_ 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
```

También se permiten determinados rangos de nombres de carácter universal en un identificador. Un nombre de carácter universal en un identificador no puede designar un carácter de control ni un carácter en el juego de caracteres de origen básico. Para obtener más información, vea [Character Sets](#). Estos rangos de números de punto de código Unicode son válidos como nombres de carácter universal para cualquier carácter de un identificador:

- 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

Los siguientes caracteres son válidos para cualquier carácter de un identificador excepto el primero:

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

Estos rangos de números de punto de código Unicode también son válidos como nombres de carácter universal para cualquier carácter de un identificador, excepto el primero:

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

Específicos de Microsoft

Solo los 2048 primeros caracteres de los identificadores de Microsoft C++ son significativos. El compilador hace que los nombres de los tipos definidos por el usuario sean "representativos" para conservar la información de tipo. El nombre resultante, incluida la información de tipo, no puede tener más de 2048 caracteres. (Consulte [Nombres representativos](#) para obtener más información). Los factores que pueden influir en la longitud de un identificador representativo son:

- Si el identificador indica un objeto de un tipo definido por el usuario o un tipo derivado de un tipo definido por el usuario.
- Si el identificador denota una función o un tipo derivado de una función.
- El número de argumentos para una función.

El signo de dólar \$ es un carácter de identificador válido en el compilador de Microsoft C++ (MSVC). MSVC también permite usar los caracteres reales representados por los rangos válidos de nombres de carácter universal en los identificadores. Para usar dichos caracteres, se debe guardar el archivo mediante una página de códigos para codificación de archivos que los incluya. En este ejemplo muestra cómo dos caracteres extendidos y nombres de carácter universal se pueden usar indistintamente en el código.

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  
}
```

El rango de caracteres permitidos en un identificador es menos restrictivo cuando se compila código C++/CLI. Los identificadores del código compilado con /clr deben regirse por el [Estándar ECMA-335: Common Language Infrastructure \(CLI\)](#).

FIN de Específicos de Microsoft

El primer carácter de un identificador debe ser un carácter alfabético, en mayúsculas o minúsculas, o un carácter de subrayado (_). Debido a que los identificadores de C++ distinguen entre mayúsculas y minúsculas, `fileName` es diferente de `FileName`.

Los identificadores no pueden escribirse igual ni presentar el mismo uso de mayúsculas y minúsculas que las palabras clave. Los identificadores que contienen palabras clave son válidos. Por ejemplo, `Pint` es un identificador válido aunque contenga `int`, que es una palabra clave.

El uso de dos caracteres de subrayado secuenciales (__) de un identificador o un único carácter de subrayado inicial seguido de una letra mayúscula se reserva para las implementaciones de C++ en todos los ámbitos. Evite el uso de un carácter de subrayado inicial seguido de una letra minúscula en los nombres con ámbito de archivo a fin de evitar posibles conflictos con los identificadores reservados actuales o futuros.

Consulte también

[Convenciones léxicas](#)

Palabras clave (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

Las palabras clave son identificadores reservados predefinidos que tienen un significado especial para el compilador. No se pueden usar como identificadores en el programa.

Las palabras clave siguientes están reservadas para Microsoft C++. Los nombres con subrayado inicial y nombres especificados para C++/CX y C++/CLI son extensiones de Microsoft.

Palabras clave de Estándar C++

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
declaración using
directiva using
virtual
void
volatile
wchar_t
while
xor^b
xor_eq^b

^a la palabra clave específica de Microsoft `__asm` reemplaza la sintaxis `asm` de C++. `asm` está reservada por compatibilidad con otras implementaciones de C++, pero no se implementa. Use `__asm` para el ensamblado insertado en destinos x86. Microsoft C++ no admite el ensamblado insertado para otros destinos.

^b Los sinónimos del operador extendido son palabras clave cuando se especifica `/permissive-` o `/Za` ([Deshabilitar extensiones de lenguaje](#)). No son palabras clave cuando están habilitadas las extensiones de Microsoft.

^c se admite cuando `/std:c++20` o posterior (por ejemplo, `/std:c++latest`) se especifica.

Palabras clave específicas de Microsoft C++

En C++, los identificadores que contienen dos guiones bajos consecutivos se reservan para las implementaciones del compilador. La convención de Microsoft es que las palabras clave específicas de Microsoft vayan precedidas por subrayados dobles. Estas palabras no se pueden utilizar como nombres de identificador.

Las extensiones de Microsoft están habilitadas de manera predeterminada. Para asegurarse de que los programas sean totalmente portables, se pueden deshabilitar las extensiones de Microsoft al especificar la opción [/permissive-](#) o [/Za\(Deshabilitar las extensiones del lenguaje\)](#) durante la compilación. Estas opciones deshabilitan algunas palabras clave específicas de Microsoft.

Con las extensiones de Microsoft habilitadas, puede usar las palabras clave específicas de Microsoft en los programas. Para la conformidad con ANSI, estas palabras clave van precedidas por un subrayado doble. Por compatibilidad con versiones anteriores, se admiten las versiones de un solo subrayado de muchas de las palabras clave con doble subrayado. La palabra clave `_cdecl` está disponible sin subrayado inicial.

La palabra clave `_asm` reemplaza la sintaxis `asm` de C++. `asm` está reservada por compatibilidad con otras implementaciones de C++, pero no se implementa. Mediante `_asm`.

La palabra clave `_based` tiene usos limitados para las compilaciones de destino de 32 y 64 bits.

[_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](#)

`_int64e`
`_int8e`
`_interface`
`_leavee`
`_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 función intrínseca que se usa en el control de eventos.

^e para la compatibilidad con versiones anteriores, estas palabras clave están disponibles tanto con dos caracteres de subrayado iniciales como con un único carácter de subrayado inicial cuando se habilitan las extensiones de Microsoft (el predeterminado).

Palabras clave de Microsoft en modificadores `_declspec`

Estos identificadores son atributos extendidos para el modificador `_declspec`. Se consideran palabras clave dentro de ese contexto.

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

Palabras clave de C++/CLI y C++/CX

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

`_try_castf`
`_valuef`
`abstractg`
`arrayg`
`as_friend`
`delegateg`
`enum class`
`enum struct`
`eventg`

`finally`
`for each in`
`gcnewg`
`genericg`
`initonly`
`interface classg`
`interface structg`
`interior_ptrg`
`literalg`

`newg`
`propertyg`
`ref class`
`ref struct`
`safecast`
`sealedg`
`typeid`
`value classg`
`value structg`

^f Aplicable solamente a Extensiones administradas para C++. Esta sintaxis ahora está en desuso. Para obtener más información, vea [Extensiones de componentes para plataformas de tiempo de ejecución](#).

^g No es aplicable a C++/CLI.

Consulte también

[Convenciones léxicas](#)

[Operadores integrados de C++, precedencia y asociatividad](#)

Signos de puntuación (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Los signos de puntuación en C++ tienen un significado sintáctico y semántico para el compilador, pero, por sí mismos, no especifican una operación que produzca un valor. Algunos signos de puntuación, solos o combinados, también pueden ser operadores de C++ o ser significativos para el preprocesador.

Cualquiera de los siguientes caracteres se consideran signos de puntuación:

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

Los signos de puntuación [], () y { } deben aparecer en parejas después de la [fase de traducción 4](#).

Consulte también

[Convenciones léxicas](#)

Literales numéricos, booleanos y de puntero

Artículo • 03/03/2023 • Tiempo de lectura: 5 minutos

Un literal es un elemento de programa que representa directamente un valor. En este artículo se tratan los literales de tipo entero, de punto flotante, booleanos y de puntero. Para obtener información sobre los literales de cadena y carácter, consulte [Literales de cadena y carácter \(C++\)](#). También puede definir sus propios literales basándose en cualquiera de estas categorías. Para obtener más información, consulte [Literales definidos por el usuario \(C++\)](#).

Puede usar literales en muchos contextos, pero lo más común es utilizarlos para inicializar variables con nombre y pasar argumentos a funciones:

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
```

A veces es importante indicar al compilador cómo interpretar un literal o qué tipo específico debe darle. Esto se logra anexando prefijos o sufijos al literal. Por ejemplo, el prefijo `0x` indica al compilador que interprete el número que sigue como un valor hexadecimal, por ejemplo, `0x35`. El sufijo `ULL` indica al compilador que trate el valor como un tipo `unsigned long long`, como en `5894345ULL`. Consulte las siguientes secciones para obtener una lista completa de los prefijos y sufijos para cada tipo de literal.

Literales enteros

Los literales enteros comienzan con un dígito y no tienen partes fraccionarias ni exponentes. Se pueden especificar en formato decimal, binario, octal o hexadecimal. Opcionalmente, puede especificar un literal entero como sin signo, y como un tipo `long` o `long long` mediante un sufijo.

Cuando no haya ningún prefijo o sufijo, el compilador proporcionará un tipo de valor literal integral de `int` (32 bits). Si el valor no se ajusta, asignará uno del tipo `long long` (64 bits).

Para especificar un literal entero decimal, comience la especificación por un dígito distinto de cero. Por ejemplo:

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
```

Para especificar un literal entero octal, comience la especificación por 0, seguido de una secuencia de dígitos entre 0 y 7. Los dígitos 8 y 9 se consideran errores al especificar un literal octal. Por ejemplo:

C++

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

Para especificar un literal entero hexadecimal, inicie la especificación con `0x` o `0X` (no importa si la "x" está en mayúsculas o minúsculas), seguido de una secuencia de dígitos entre 0 y 9 y de a (o A) a f (o F). Los dígitos hexadecimales de a (o A) a f (o F) representan valores comprendidos en el intervalo de 10 a 15. Por ejemplo:

C++

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

Para especificar un tipo sin signo, use el sufijo `u` o `U`. Para especificar un tipo long, use el sufijo `l` o `L`. Para especificar un tipo entero de 64 bits, utilice el sufijo `LL` o `ll`. El sufijo `i64` sigue siendo compatible, pero no se recomienda. Es específico de Microsoft y no es portable. Por ejemplo:

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
```

Separadores de dígitos: puede usar el carácter de comilla simple (apóstrofo) para separar los valores de contexto de los números más grandes para facilitar su lectura. Los separadores no afectan a la compilación.

C++

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

Literales de punto flotante

Los literales de punto flotante especifican los valores que debe tener una parte fraccionaria. Estos valores contienen separadores decimales (.) y pueden incluir exponentes.

Los literales de punto flotante tienen un *significado* (a veces denominado *mantisa*), que especifica el valor del número. También tienen un *exponente*, que especifica la magnitud del número, y un sufijo opcional que especifica el tipo de literal. El significando se especifica como una secuencia de dígitos seguidos de un punto y de una secuencia opcional de dígitos que representan la parte fraccionaria del número. Por ejemplo:

C++

```
18.46  
38.
```

El exponente, si está presente, especifica la magnitud del número como potencia de 10, tal como se muestra en el ejemplo siguiente:

C++

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

El exponente se puede especificar mediante e o E, que tienen el mismo significado, seguido de un signo opcional (+ o -) y una secuencia de dígitos. Si un exponente está presente, el separador decimal final es innecesario en los números enteros como 18E0.

Los literales de punto flotante tienen el tipo `double` de forma predeterminada. Mediante los sufijos f o l (o F o L, ya que el sufijo no distingue entre mayúsculas y minúsculas), el literal se puede especificar como `float` o `long double`.

Aunque `long double` y `double` tienen la misma representación, no son el mismo tipo.

Por ejemplo, puede haber funciones sobrecargadas como:

C++

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

y

C++

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

booleanos, literales

Los literales booleanos son `true` y `false`.

Literal de puntero (C++11)

C++ introduce el literal `nullptr` para especificar un puntero inicializado a cero. En código portable, debe usarse `nullptr` en lugar de cero de tipo entero o macros como `NULL`.

Literales binarios (C++14)

Un literal binario puede especificarse mediante el uso del prefijo `0B` o `0b` seguido por una secuencia de unos y ceros:

C++

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

Evite usar literales como “constantes mágicas”

Puede usar literales directamente en expresiones e instrucciones, aunque no siempre es una buena práctica de programación:

C++

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

En el ejemplo anterior, es mejor usar una constante con nombre que tenga un significado claro, como "MAXIMUM_ERROR_THRESHOLD". Y si los usuarios finales ven el valor devuelto "Success" (Correcto), sería mejor usar una constante de cadena con nombre. Las constantes de cadena pueden almacenarse en una sola ubicación en un archivo desde el que se pueda localizar a otros idiomas. Además, usar constantes con nombre ayuda a comprender el objetivo del código a otras personas y a usted mismo.

Consulte también

[Convenciones léxicas](#)

[Literales de cadena de C++](#)

[Literales de C++ definidos por el usuario](#)

Literales de cadena y carácter (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 21 minutos

C++ admite varios tipos de cadenas y caracteres, y proporciona maneras de expresar valores literales de cada uno de esos tipos. En el código fuente, el contenido de los literales de carácter y cadena se expresa mediante un juego de caracteres. Los nombres de carácter universal y los caracteres de escape permiten expresar cualquier cadena con tan solo el juego básico de caracteres de código fuente. Un literal de cadena sin formato permite evitar la utilización de caracteres de escape y puede usarse para expresar todos los tipos de literales de cadena. También se pueden crear literales `std::string` sin necesidad de seguir pasos adicionales de construcción o conversión.

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
}

```

Los literales de cadena no tienen prefijos o tienen los prefijos `u8`, `L`, `u` y `U` para denotar caracteres estrechos (byte único o multibyte), UTF-8, caracteres anchos (UCS-2 o UTF-16), codificaciones UTF-16 y UTF-32, respectivamente. Un literal de cadena sin formato puede tener los prefijos `R`, `u8R`, `LR`, `uR` y `UR` para los equivalentes sin formato de estas codificaciones. Para crear valores `std::string` temporales o estáticos, puede usar literales de cadena o literales de cadena sin formato con un sufijo `s`. Para obtener más información, consulte la sección [Literales de cadena](#) a continuación. Para obtener más información sobre el juego básico de caracteres de código fuente, los nombres de carácter universal y el uso de caracteres para páginas de códigos extendidas en el código fuente, consulte [Juego de caracteres](#).

Literales de carácter

Un *literal de carácter* está compuesto por un carácter de constante. Se representa mediante el carácter delimitado por comillas simples. Hay cinco tipos de literales de carácter:

- Literales de caracteres ordinarios de tipo `char` (por ejemplo, `'a'`)
- Literales de caracteres UTF-8 de tipo `char` (`char8_t` en C++20), por ejemplo `u8'a'`
- Literales de caracteres anchos de tipo `wchar_t` (por ejemplo, `L'a'`)
- Literales de caracteres UTF-16 de tipo `char16_t` (por ejemplo, `u'a'`)
- Literales de caracteres UTF-32 de tipo `char32_t` (por ejemplo, `U'a'`)

El carácter que se usa para un literal de carácter puede ser cualquier carácter, a excepción de los caracteres reservados como la barra diagonal inversa (\), las comillas simples (') o la nueva línea. Los caracteres reservados se pueden especificar mediante una secuencia de escape. Los caracteres se pueden especificar mediante nombres de carácter universal, siempre que el tipo sea lo suficientemente grande como para contener al carácter.

Encoding

Los literales de caracteres se codifican de forma diferente en función de su prefijo.

- Un literal de carácter sin prefijo es un literal de carácter normal. El valor de un literal de carácter normal que contiene un solo carácter, una secuencia de escape o un nombre de carácter universal que se puede representar en el juego de caracteres de ejecución tiene un valor igual al valor numérico de su codificación en el juego de caracteres de ejecución. Un literal de carácter normal que contiene más de un carácter, una secuencia de escape o un nombre de carácter universal es un *literal multicaracter*. Un literal multicaracter o un literal de carácter normal que no se puede representar en el juego de caracteres de ejecución tiene el tipo `int` y su valor está definido por la implementación. Para MSVC, consulte la sección **específica de Microsoft** más adelante.
- Un literal de carácter que comienza con el prefijo L es un literal de caracteres anchos. El valor de un literal de caracteres anchos que contiene un solo carácter, secuencia de escape o nombre de carácter universal tiene un valor igual al valor numérico de su codificación en el juego de caracteres anchos de ejecución a menos que el literal de caracteres no tenga ninguna representación en el juego de caracteres anchos de ejecución, en cuyo caso el valor está definido por la implementación. El valor de un literal de caracteres anchos que contiene varios caracteres, secuencias de escape o nombres de caracteres universales está definido por la implementación. Para MSVC, consulte la sección **específica de Microsoft** más adelante.
- Un literal de carácter que comienza con el prefijo u8 es un literal de caracteres UTF-8. El valor de un literal de caracteres UTF-8 que contiene un solo carácter, secuencia de escape o nombre de carácter universal tiene un valor igual a su valor de punto de código ISO 10646 si se puede representar mediante una sola unidad de código UTF-8 (correspondiente al bloque C0 Controls y Basic Latin Unicode). Si el valor no se puede representar mediante una sola unidad de código UTF-8, el programa tiene un formato incorrecto. Un literal de caracteres UTF-8 que contiene

más de un carácter, secuencia de escape o nombre de carácter universal no tiene formato.

- Un literal de carácter que comienza con el prefijo `u` es un literal de caracteres UTF-16. El valor de un literal de caracteres UTF-16 que contiene un solo carácter, secuencia de escape o nombre de carácter universal tiene un valor igual a su valor de punto de código ISO 10646 si se puede representar mediante una sola unidad de código UTF-16 (correspondiente al plano multilingüe básico). Si el valor no se puede representar mediante una sola unidad de código UTF-16, el programa tiene un formato incorrecto. Un literal de caracteres UTF-16 que contiene más de un carácter, secuencia de escape o nombre de carácter universal no tiene formato.
- Un literal de carácter que comienza con el prefijo `U` es un literal de caracteres UTF-32. El valor de un literal de caracteres UTF-32 que contiene un solo carácter, secuencia de escape o nombre de carácter universal tiene un valor igual a su valor de punto de código ISO 10646. Un literal de caracteres UTF-32 que contiene más de un carácter, secuencia de escape o nombre de carácter universal no tiene formato.

Secuencias de escape

Hay tres tipos de secuencias de escape: simple, octal, hexadecimal. Las secuencias de escape pueden ser cualquiera de los siguientes valores:

| Value | Secuencia de escape |
|------------------------|---------------------|
| Nueva línea | \n |
| Barra diagonal inversa | \\\ |
| Tabulación horizontal | \t |
| interrogación | ? o \? |
| Tabulación vertical | \v |
| Comilla simple | \' |
| retroceso | \b |
| Comilla doble | \" |
| retorno de carro | \r |
| Carácter nulo | \0 |
| avance de página | \f |

| Value | Secuencia de escape |
|------------------|---------------------|
| Octal | \ooo |
| Alerta (campana) | \a |
| Hexadecimal | \xhhh |

Una secuencia de escape octal es una barra diagonal inversa seguida de una secuencia de uno a tres dígitos octales. Una secuencia de escape octal finaliza en el primer carácter que no es un dígito octal, si se encuentra antes que el tercer dígito. El valor octal más alto posible es \377.

Una secuencia de escape hexadecimal es una barra diagonal inversa seguida del carácter x y seguida de una secuencia de uno o varios dígitos hexadecimales. Los ceros a la izquierda se ignoran. En un literal de carácter estrecho sin prefijo o con prefijo u8, el valor hexadecimal máximo es 0xFF. En un literal de carácter ancho con prefijo L o prefijo u, el valor hexadecimal máximo es 0xFFFF. En un literal de carácter ancho con prefijo U, el valor hexadecimal máximo es 0xFFFFFFFF.

En este código de ejemplo se muestran algunos ejemplos de caracteres de escape mediante literales de caracteres normales. La misma sintaxis de secuencia de escape es válida para los demás tipos literales de caracteres.

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
*/
```

El carácter de barra diagonal inversa (\) es un carácter de continuación de línea cuando se coloca al final de una línea. Si quiere que un carácter de barra diagonal inversa aparezca como un literal de carácter, debe escribir dos barras diagonales inversas en una fila (\ \). Para obtener más información sobre el carácter de continuación de línea, consulte [Phases of Translation](#).

Específico de Microsoft

Para crear un valor a partir de un literal multicaracter estrecho, el compilador convierte el carácter o la secuencia de caracteres entre comillas simples en valores de ocho bits dentro de un entero de 32 bits. Varios caracteres del literal llenan los bytes correspondientes según sea necesario de orden superior a orden inferior. A continuación, el compilador convierte el entero en el tipo de destino siguiendo las reglas habituales. Por ejemplo, para crear un valor `char`, el compilador usa el byte de orden inferior. Para crear un valor `wchar_t` o `char16_t`, el compilador usa la palabra de orden inferior. El compilador advierte que el resultado se trunca si cualquiera de los bits se establece por encima del byte o la palabra asignados.

C++

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

Una secuencia de escape octal que parece contener más de tres dígitos se trata como una secuencia octal de 3 dígitos, seguida de los dígitos posteriores como caracteres en un literal multicaracter, lo que puede dar resultados sorprendentes. Por ejemplo:

C++

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

Las secuencias de escape que parecen contener caracteres que no son octales se evalúan como una secuencia octal hasta el último carácter octal, seguido del resto de los caracteres como caracteres posteriores en un literal multicaracter. Advertencia C4125 se genera si el primer carácter no octal es un dígito decimal. Por ejemplo:

C++

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

Secuencia de escape octal que tiene un valor mayor que \377 causa el error C2022: '*value-in-decimal*': demasiado grande para el carácter.

Una secuencia de escape que parece tener caracteres hexadecimales y no hexadecimales se evalúa como un literal multícarácter que contiene una secuencia de escape hexadecimal hasta el último carácter hexadecimal, seguido de los caracteres no hexadecimales. Una secuencia de escape hexadecimal que no contiene ningún dígito hexadecimal produce el error del compilador C2153: "los literales hexadecimales deben tener al menos un dígito hexadecimal".

C++

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

Si un literal de caracteres anchos con prefijo L contiene una secuencia de varios caracteres, el valor se toma del primer carácter y el compilador genera la advertencia C4066. Los caracteres subsiguientes se ignoran, a diferencia del comportamiento que se observa en el literal multícarácter ordinario equivalente.

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
```

La sección específica de Microsoft termina aquí.

nombres de carácter universal

En los literales de carácter y los literales de cadena nativa (con formato), se puede representar cualquier carácter mediante un nombre de carácter universal. Los nombres de carácter universal se forman con un prefijo \u seguido de un punto de código Unicode de ocho dígitos o con un prefijo \u seguido de un punto de código Unicode

de cuatro dígitos. La totalidad de los dígitos (ocho o cuatro, respectivamente) debe estar presente para formar un nombre de carácter universal correctamente.

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'
```

Pares suplentes

Los nombres de carácter universal no pueden codificar valores que se encuentran en el rango de punto de código suplente de D800 a DFFF. En el caso de pares suplentes Unicode, especifique el nombre de carácter universal mediante `\UNNNNNNNN`, donde NNNNNNNN es el punto de código de ocho dígitos para el carácter. El compilador genera un par suplente si es necesario.

En C++03, el lenguaje solo permitía representar un subconjunto de caracteres por sus nombres de caracteres universales y permitía algunos nombres de caracteres universales que no representaran realmente caracteres Unicode válidos. Este error se corrigió en el estándar de C++11. En C++11, tanto los literales de carácter y cadena como los identificadores pueden usar nombres de carácter universal. Para obtener más información sobre los nombres de carácter universal, consulte [Character Sets](#). Para obtener más información sobre Unicode, consulte [Unicode](#). Para obtener más información sobre los pares suplentes, consulte [Pares suplentes y caracteres complementarios](#).

Literales de cadena

Un literal de cadena representa una secuencia de caracteres que, en conjunto, forman una cadena terminada en null. Los caracteres deben escribirse entre comillas. Hay los siguientes tipos de literales de cadena:

Literales de cadena estrechos

Un literal de cadena estrecho es una matriz sin prefijo, delimitada por comillas dobles y terminada en null del tipo `const char[n]`, donde n es la longitud de la matriz en bytes. Un literal de cadena estrecho puede contener cualquier carácter gráfico, excepto las comillas dobles (""), la barra diagonal inversa (\) o el carácter de nueva línea (\n). Un

literal de cadena estrecho también puede contener las secuencias de escape antes mencionadas, así como nombres de carácter universal que caben en un byte.

C++

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

Cadenas con codificación UTF-8

Una cadena con codificación UTF-8 es una matriz con prefijo u8, delimitada por comillas dobles y terminada en null del tipo `const char[n]`, donde *n* es la longitud de la matriz codificada en bytes. Un literal de cadena con prefijo u8 puede tener cualquier carácter gráfico, excepto las comillas dobles ("), la barra diagonal inversa (\) o el carácter de nueva línea (\n). También puede contener las secuencias de escape de secuencias antes mencionadas y cualquier nombre de carácter universal.

C++20 presenta el tipo de carácter portátil `char8_t` (Unicode codificado con UTF-8) de 8 bits. En C++20, los prefijos literales u8 especifican caracteres o cadenas de `char8_t` en lugar de `char`.

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:-)";
```

Literales de cadena anchos

Un literal de cadena ancho es una matriz terminada en null de valores `wchar_t` constantes que lleva el prefijo 'L' y que contiene cualquier carácter gráfico excepto las comillas dobles ("), la barra diagonal inversa (\) o el carácter de nueva línea. También puede contener las secuencias de escape de secuencias antes mencionadas y cualquier nombre de carácter universal.

C++

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

char16_t y char32_t (C++11)

C++ 11 incluye los tipos de caracteres portables `char16_t` (Unicode de 16 bits) y `char32_t` (32 bits Unicode):

C++

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

Literales de cadena sin formato (C++11)

Un literal de cadena sin formato es una matriz terminada en null (de cualquier tipo de carácter) que contiene cualquier carácter gráfico, incluidas las comillas dobles ("), la barra diagonal inversa (\) o el carácter de nueva línea. Los literales de cadena sin formato suelen usarse en expresiones regulares que utilizan clases de caracteres, y en las cadenas HTML y XML. Para obtener ejemplos, vea el siguiente artículo: [preguntas más frecuentes de Bjarne Stroustrup sobre C++11](#).

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);
```

Un delimitador es una secuencia definida por el usuario de hasta 16 caracteres que precede inmediatamente al paréntesis de apertura de un literal de cadena sin formato, y sigue inmediatamente a su paréntesis de cierre. Por ejemplo, en `R"abc(Hello"\()abc"` la secuencia de delimitador es `abc` y el contenido de la cadena es `Hello"\()`. Puede usar un delimitador para eliminar la ambigüedad de las cadenas sin formato que contienen comillas dobles y paréntesis. Este literal de cadena produce un error del compilador:

C++

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

Pero un delimitador lo resuelve:

C++

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

Puede construir un literal de cadena sin formato que contiene una línea nueva (no el carácter de escape) en el código fuente:

C++

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

Literales std::string (C++14)

Los literales `std::string` son implementaciones de la biblioteca estándar de literales definidos por el usuario (véase abajo) que se representan como `"xyz"s` (con un sufijo `s`). Este tipo de literal de cadena produce un objeto temporal de tipo `std::string`, `std::wstring`, `std::u32string` o `std::u16string` según el prefijo que se especifique. Cuando no se usa ningún prefijo, como en el caso de arriba, se genera un `std::string`. `L"xyz"s` genera un `std::wstring`. `u"xyz"s` produce un `std::u16string` y `U"xyz"s` produce un `std::u32string`.

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 };
```

El sufijo `s` también puede usarse en literales de cadena sin formato:

C++

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

Los literales `std::string` se definen en el espacio de nombres

`std::literals::string_literals` del archivo de encabezado `<string>`. Dado que tanto `std::literals::string_literals` como `std::literals` se declaran como [espacios de nombres alineados](#), `std::literals::string_literals` se trata automáticamente como si perteneciera directamente al espacio de nombres `std`.

Tamaño de literales de cadena

En cadenas `char*` de ANSI y otras codificaciones de byte único (pero no UTF-8), el tamaño (en bytes) de un literal de cadena es el número de caracteres más uno (para el carácter nulo de terminación). En el resto de los tipos de cadena, el tamaño no está estrictamente relacionado con el número de caracteres. UTF-8 usa hasta cuatro elementos `char` para codificar algunas *unidades de código*; `char16_t` o `wchar_t` codificados como UTF-16 pueden usar dos elementos (para un total de cuatro bytes) para codificar una única *unidad de código*. En este ejemplo se muestra el tamaño, en bytes, de un literal de cadena ancho:

C++

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

Observe que `strlen()` y `wcslen()` no incluyen el tamaño del carácter nulo de terminación, cuyo tamaño es igual que el tamaño del elemento del tipo `string`: un byte en una cadena de `char*` o `char8_t*`, dos bytes en cadenas `wchar_t*` o `char16_t*` y cuatro bytes en cadenas de `char32_t*`.

En versiones anteriores a Visual Studio 2022, versión 17.0, la longitud máxima de un literal de cadena es de 65 535 bytes. Este límite se aplica a los literales de cadena estrechos y anchos. En Visual Studio 2022, versión 17.0 y posteriores, esta restricción se suprime y la longitud de cadena la limitan los recursos disponibles.

Modificación de literales de cadena

Dado que los literales de cadena son constantes (a excepción de los literales `std::string`), intentar modificarlos (por ejemplo, `str[2] = 'A'`) provoca un error del compilador.

Específico de Microsoft

En Microsoft C++, puede utilizar un literal de cadena para inicializar un puntero a `char` o a `wchar_t` no constantes. Esta inicialización no const se permite en el código C99, pero está en desuso en C++98 y se eliminó en C++11. Un intento de modificar la cadena produce una infracción de acceso, como en este ejemplo:

C++

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

Puede hacer que el compilador emita un error cuando un literal de cadena se convierte en un puntero de carácter non-const al establecer la opción del compilador [/Zc:strictStrings\(Disable string literal type conversion\)](#). Es recomendable para el código portable que cumple los estándares. También es conveniente usar la palabra clave `auto` para declarar punteros inicializados de literales de cadena, porque resuelve el tipo (const) correcto. Por ejemplo, en este ejemplo de código se detecta un intento de escribir en un literal de cadena en tiempo de compilación:

C++

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

En algunos casos, pueden agruparse literales de cadena idénticos para ahorrar espacio en el archivo ejecutable. En la agrupación de literales de cadena, el compilador hace que todas las referencias a un literal de cadena determinado apunten a la misma ubicación de la memoria, en lugar de apuntar cada una a una instancia distinta del literal de cadena. Para habilitar la agrupación de cadenas, use la opción del compilador [/GF](#).

La sección específica de Microsoft termina aquí.

Concatenación de literales de cadena adyacentes

Los literales de cadena anchos o estrechos adyacentes se concatenan. Esta declaración:

C++

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

es idéntica a esta declaración:

```
C++
```

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

y a esta declaración:

```
C++
```

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

El uso de códigos de escape hexadecimales insertados para especificar literales de cadena puede producir resultados inesperados. El ejemplo siguiente está pensado para crear un literal de cadena que contiene el carácter ASCII 5, seguido de los caracteres f, i, v y e:

```
C++
```

```
"\x05five"
```

El resultado real es 5F hexadecimal, que es el código ASCII de un carácter de subrayado, seguido de los caracteres i, v y e. Para obtener el resultado correcto, puede utilizar una de estas secuencia de escape:

```
C++
```

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

Los literales `std::string` (y los relacionados `std::u8string`, `std::u16string` y `std::u32string`) se pueden concatenar con el operador `+` definido para los tipos `basic_string`. También pueden concatenarse de la misma manera que literales de cadena adyacentes. En ambos casos, la codificación de cadena y el sufijo deben coincidir:

```
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
```

Literales de cadena con nombres de caracteres universales

Los literales de cadena nativos (con formato) pueden usar nombres de carácter universal para representar cualquier carácter, siempre que el nombre de carácter universal pueda codificarse como uno o varios caracteres en el tipo de cadena. Por ejemplo, no se puede codificar un nombre de carácter universal que representa un carácter extendido en una cadena de caracteres estrechos mediante la página de códigos ANSI, pero puede codificarse en cadenas de caracteres estrechos de algunas páginas de códigos multibyte, así como en cadenas UTF-8 o en una cadena de caracteres anchos. En C++11, los tipos de cadena `char16_t*` y `char32_t*` amplían la compatibilidad con Unicode y C++20 lo extiende al tipo `char8_t`:

C++

```
// ASCII smiling face
const char*      s1 = ":-)";

// UTF-16 (on Windows) encoded WINKING FACE (U+1F609)
const wchar_t*   s2 = L"\ud83d\udc99 is ;-)";

// UTF-8 encoded SMILING FACE WITH HALO (U+1F607)
const char*      s3a = u8"\ud83d\udc99 is \ud83d\udc99 is 0:-)"; // Before C++20
const char8_t*   s3b = u8"\ud83d\udc99 = \ud83d\udc99 is 0:-)"; // C++20

// UTF-16 encoded SMILING FACE WITH OPEN MOUTH (U+1F603)
const char16_t*  s4 = u"\ud83d\udc99 = \ud83d\udc99 is :-D";

// UTF-32 encoded SMILING FACE WITH SUNGLASSES (U+1F60E)
const char32_t*  s5 = U"\ud83d\udc99 = \ud83d\udc99 is B-)";
```

Consulte también

- [Juegos de caracteres](#)
- [Literales numéricos, booleanos y de puntero](#)
- [Literales definidos por el usuario](#)

Literales definidos por el usuario

Artículo • 03/03/2023 • Tiempo de lectura: 6 minutos

Hay seis categorías principales de literales en C++: entero, carácter, punto flotante, cadena, booleano y puntero. A partir de C++ 11, puede definir sus propios literales en función de estas categorías para proporcionar atajos sintácticos para expresiones comunes y aumentar la seguridad de tipos. Por ejemplo, supongamos que tiene una clase `Distance`. Puede definir un literal para kilómetros y otro para millas, y fomentar que el usuario sea explícito sobre las unidades de medida al escribir: `auto d = 42.0_km` o `auto d = 42.0_mi`. No hay ninguna ventaja ni desventaja de rendimiento con literales definidos por el usuario; se usan principalmente por comodidad o para la deducción de tipos de tiempo de compilación. La biblioteca estándar tiene literales definidos por el usuario para `std::string`, `std::complex` y unidades de operaciones de tiempo y duración en el encabezado `<chrono>`:

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
```

Firmas de operador literal definido por el usuario

Implemente un literal definido por el usuario mediante la definición de un `operator""` en el ámbito de espacio de nombres de una de las siguientes formas:

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

```

Los nombres de operador del ejemplo anterior son marcadores de posición para cualquier nombre que proporcione; sin embargo, el carácter de subrayado inicial es necesario. (Solo se permite que la biblioteca estándar defina literales sin el carácter de subrayado). El tipo de valor devuelto es donde se personaliza la conversión u otras operaciones realizadas por el literal. Además, cualquiera de estos operadores se puede definir como `constexpr`.

Literales elaborados

En el código fuente cualquier literal, definido por el usuario o no, es esencialmente una secuencia de caracteres alfanuméricos, como `101`, `54.7`, `"hello"` o `true`. El compilador interpreta la secuencia como integer, float, const char * string, etc. Un valor literal definido por el usuario que acepta como entrada cualquier tipo que el compilador asigne al valor literal se conoce informalmente como un *literal elaborado*. Todos los operadores anteriores, excepto `_r` y `_t`, son literales elaborados. Por ejemplo, un literal `42.0_km` se enlazaría a un operador denominado `_km` que tuviera una firma semejante a `_b` y el literal `42_km` se enlazaría a un operador con una firma semejante a `_a`.

En el ejemplo siguiente se muestra cómo los literales definidos por el usuario pueden fomentar que los llamadores sean explícitos sobre los datos proporcionados. Para crear un `Distance`, el usuario debe especificar explícitamente kilómetros o millas mediante el literal definido por el usuario adecuado. Se puede lograr el mismo resultado de otras maneras, pero los literales definidos por el usuario son menos detallados que las alternativas.

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;
}

```

El número literal debe usar un decimal. De lo contrario, el número se interpretaría como un número entero y el tipo no sería compatible con el operador. Para la entrada de punto flotante, el tipo debe ser `long double`, y para los tipos enteros debe ser `long long`.

Literales sin formato

En un literal definido por el usuario sin formato, el operador que defina aceptará el literal como secuencia de valores de caracteres. Es necesario interpretar esa secuencia como un número o una cadena u otro tipo. En la lista de operadores mostrada anteriormente en esta página, se pueden usar `_r` y `_t` para definir literales sin formato:

C++

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

Puede usar literales sin formato para proporcionar una interpretación personalizada de una secuencia de entrada que sea diferente del comportamiento normal del compilador. Por ejemplo, puede definir un literal que convierta la secuencia `4.75987` en un tipo Decimal personalizado en lugar de un tipo de punto flotante de IEEE 754. Los literales sin formato, como los literales elaborados, pueden usarse también para la validación en tiempo de compilación de las secuencias de entrada.

Ejemplo: Limitaciones de literales sin formato

El operador literal sin formato y la plantilla de operador literal solo funcionan para literales de entero y de punto flotante definidos por el usuario, tal y como se muestra en el ejemplo siguiente:

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<===
```

Conceptos básicos (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

En esta sección se explican conceptos necesarios para entender C++. Los programadores de C estarán familiarizados con muchos de estos conceptos, pero hay algunas diferencias sutiles que pueden producir resultados inesperados del programa. Se tratan los siguientes temas:

- [Sistema de tipos de C++](#)
- [Ámbito](#)
- [Unidades de traducción y vinculación](#)
- [Función main y argumentos de la línea de comandos](#)
- [Finalización del programa](#)
- [Lvalues y Rvalues](#)
- [Objetos temporales](#)
- [Alineación](#)
- [Tipos POD, de diseño estándar y triviales](#)

Vea también

[Referencia del lenguaje C++](#)

Sistema de tipos de C++

Artículo • 03/03/2023 • Tiempo de lectura: 14 minutos

El concepto de *tipo* es importante en C++. Cada variable, argumento de función y valor devuelto por una función debe tener un tipo para compilarse. Además, el compilador asigna implícitamente un tipo a todas las expresiones (incluidos los valores literales) antes de que se evalúen. Algunos ejemplos de tipos incluyen tipos integrados, como `int` para almacenar valores enteros, `double` para almacenar valores de punto flotante o tipos de biblioteca estándar, como la clase `std::basic_string` para almacenar texto. Puede crear su propio tipo si define un objeto `class` o `struct`. El tipo especifica la cantidad de memoria asignada para la variable (o resultado de expresión). El tipo también especifica los tipos de valores que se pueden almacenar, cómo interpreta el compilador los patrones de bits en esos valores y las operaciones que puede realizar en ellos. Este artículo contiene información general sobre las principales características del sistema de tipos de C++.

Terminología

Tipo escalar: tipo que contiene un valor único de un intervalo definido. Los escalares incluyen tipos aritméticos (valores enteros o de punto flotante), miembros de tipo de enumeración, tipos de puntero, tipos de puntero a miembro y `std::nullptr_t`. Los tipos fundamentales suelen ser tipos escalares.

Tipo compuesto: tipo que no es un tipo escalar. Los tipos compuestos incluyen tipos de matriz, tipos de función, tipos de clase (o struct), tipos de unión, enumeraciones, referencias y punteros a miembros de clase no estáticos.

Variable: nombre simbólico de una cantidad de datos. El nombre se puede usar para acceder a los datos a los que hace referencia en todo el ámbito del código donde se define. En C++, la *variable* se usa a menudo para hacer referencia a instancias de tipos de datos escalares, mientras que las instancias de otros tipos normalmente se denominan *objetos*.

Objeto: para simplificar y mantener la coherencia, en este artículo se usa el término *objeto* para hacer referencia a cualquier instancia de una clase o estructura. Cuando se usa en el sentido general, incluye todos los tipos, incluso variables escalares.

Tipo POD (datos antiguos sin formato): esta categoría informal de tipos de datos de C++ hace referencia a los tipos que son escalares (consulte la sección de tipos fundamentales) o que son *clases POD*. Una clase POD no tiene miembros de datos

estáticos que tampoco son POD y no tiene constructores definidos por el usuario, destructores definidos por el usuario ni operadores de asignación definidos por el usuario. Además, las clases POD no tienen funciones virtuales, clases base ni ningún miembro de datos no estático privado o protegido. Los tipos POD suelen utilizarse para el intercambio de datos externos, por ejemplo, con un módulo escrito en lenguaje C (que solo tiene tipos POD).

Especificar tipos de variable y función

C++ es un lenguaje *fueramente tipado* y un lenguaje *con tipo estático*; cada objeto tiene un tipo y ese tipo nunca cambia. Al declarar una variable en el código, debe especificar explícitamente su tipo o usar la palabra clave `auto` para indicar al compilador que deduzca el tipo desde el inicializador. Al declarar una función en el código, debe especificar el tipo de su valor devuelto y de cada argumento. Use el tipo `void` de valor devuelto si la función no devuelve ningún valor. La excepción es cuando se usan plantillas de función, que permiten argumentos de tipos arbitrarios.

Después de declarar por primera vez una variable, no se puede cambiar su tipo en algún momento posterior. Sin embargo, puede copiar el valor devuelto de la variable o el valor devuelto de una función en otra variable de un tipo diferente. Este tipo de operaciones se denominan *conversiones de tipo*. Estas conversiones a veces resultan necesarias, aunque también pueden producir errores o pérdidas de datos.

Al declarar una variable de tipo POD, se recomienda *inicializarla*, lo que significa darle un valor inicial. Una variable, hasta que se inicializa, tiene el valor "no utilizado", que se compone de los bits que estaban previamente en esa ubicación de memoria. Es un aspecto importante de C++ recordar, especialmente si procede de otro lenguaje que controla la inicialización. Cuando se declara una variable de tipo de clase que no es POD, el constructor controla la inicialización.

En el ejemplo siguiente se muestran algunas sencillas declaraciones de variable con descripciones de cada una de ellas. En el ejemplo se muestra también cómo el compilador utiliza la información de tipo para permitir o no permitir que posteriormente se realicen ciertas operaciones en la variable.

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
```

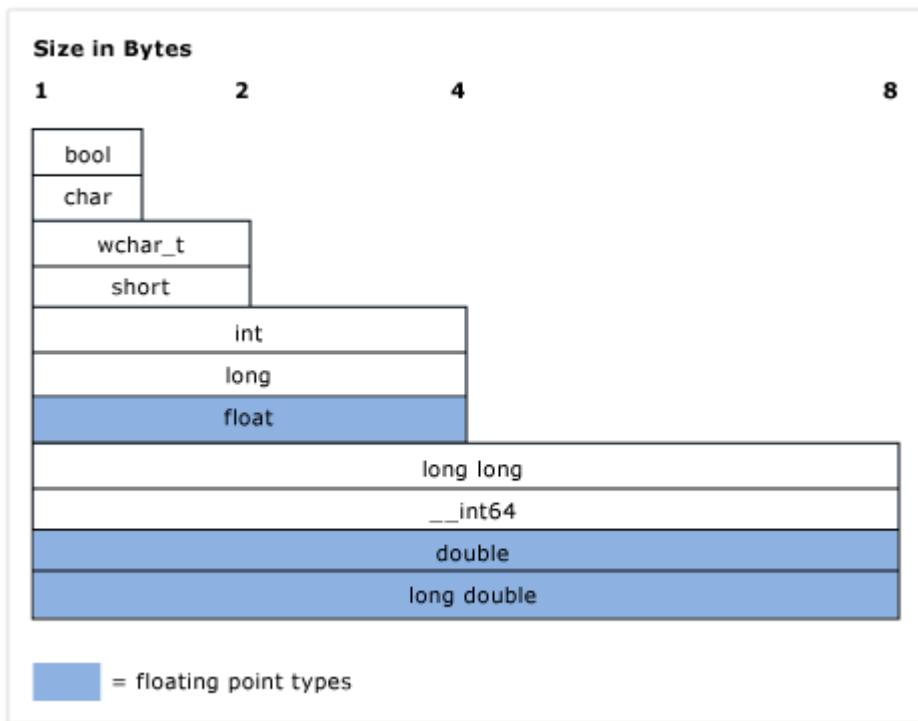
```
age = 12;                                // without an initializing value.  
                                         // error. Variable declaration must  
                                         // specify a type or use auto!  
  
result = "Kenny G.;"                      // error. Can't assign 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.
```

Tipos (integrados) fundamentales

A diferencia de algunos lenguajes, C++ no tiene un tipo base universal del que se deriven todos los demás tipos. El lenguaje contiene muchos *tipos fundamentales*, también conocidos como *tipos integrados*. Estos tipos incluyen tipos numéricos como `int`, `double`, `long`, `bool` además de los `char` tipos y `wchar_t` para los caracteres ASCII y UNICODE, respectivamente. La mayoría de los tipos fundamentales (excepto `bool`, `double`, `wchar_t` y tipos relacionados) tienen versiones `unsigned`, que modifican el intervalo de valores que la variable puede almacenar. Por ejemplo, un valor `int`, que almacena un entero de 32 bits con signo, puede representar un valor comprendido entre -2 147 483 648 y 2 147 483 647. Un `unsigned int`, que también se almacena como 32 bits, puede almacenar un valor de 0 a 4.294.967.295. El número total de valores posibles en cada caso es el mismo; solo cambia el intervalo.

El compilador reconoce estos tipos integrados y tiene reglas integradas que rigen las operaciones que se pueden realizar en ellos y cómo se pueden convertir a otros tipos fundamentales. Para obtener una lista completa de los tipos integrados y sus límites numéricos y de tamaño, consulte [Tipos integrados](#).

En la ilustración siguiente se muestran los tamaños relativos de los tipos integrados en la implementación de Microsoft C++:



En la tabla siguiente se muestran los tipos fundamentales usados con más frecuencia y sus tamaños en la implementación de Microsoft C++:

| Tipo | Size | Comentario |
|----------------------------|---------|---|
| <code>int</code> | 4 bytes | Opción predeterminada para los valores enteros. |
| <code>double</code> | 8 bytes | Opción predeterminada para los valores de punto flotante. |
| <code>bool</code> | 1 byte | Representa valores que pueden ser true o false. |
| <code>char</code> | 1 byte | Se utiliza en los caracteres ASCII de cadenas de estilo C antiguas u objetos <code>std::string</code> que nunca tendrán que convertirse a UNICODE. |
| <code>wchar_t</code> | 2 bytes | Representa valores de caracteres "anchos" que se pueden codificar en formato UNICODE (UTF-16 en Windows; puede diferir en otros sistemas operativos). <code>wchar_t</code> es el tipo de carácter que se usa en cadenas de tipo <code>std::wstring</code> . |
| <code>unsigned char</code> | 1 byte | C++ no tiene un tipo byte integrado. Use <code>unsigned char</code> para representar un valor de byte. |
| <code>unsigned int</code> | 4 bytes | Opción predeterminada para los marcadores de bits. |
| <code>long</code> | 8 bytes | Representa un intervalo mucho mayor de valores enteros. |
| <code>long</code> | bytes | |

Otras implementaciones de C++ pueden usar tamaños diferentes para determinados tipos numéricos. Para obtener más información sobre los tamaños y las relaciones de tamaño que requiere el estándar de C++, consulte [Tipos integrados](#).

El tipo `void`

El `void` tipo es un tipo especial; no se puede declarar una variable de tipo `void`, pero se puede declarar una variable de tipo `void *` (puntero a `void`), que a veces es necesaria al asignar memoria sin formato (sin tipo). Sin embargo, los punteros a `void` no son seguros para tipos y se desaconseja su uso en C++moderno. En una declaración de función, un `void` valor devuelto significa que la función no devuelve un valor; su uso como tipo de valor devuelto es un uso común y aceptable de `void`. Aunque las funciones necesarias del lenguaje C que tienen cero parámetros para declarar `void` en la lista de parámetros, por ejemplo, `fn(void)`, esta práctica no se recomienda en C++moderno; se debe declarar `fn()` una función sin parámetros . Para obtener más información, vea [Conversiones de tipos y seguridad de tipos](#).

`const` calificador de tipo

La palabra clave puede calificar cualquier tipo integrado o definido por el `const` usuario. Además, las funciones miembro pueden calificarse con `const` e incluso sobrecargarse con `const`. El valor de un `const` tipo no se puede modificar una vez inicializado.

C++

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

El `const` calificador se usa ampliamente en declaraciones de función y variable y "corrección const" es un concepto importante en C++; básicamente significa usar `const` para garantizar, en tiempo de compilación, que los valores no se modifican involuntariamente. Para obtener más información, vea [const](#).

Un `const` tipo es distinto de su versión que no `const` es ; por ejemplo, `const int` es un tipo distinto de `int`. Puede usar el operador `const_cast` de C++en las raras ocasiones en las que deba quitar la *declaración como constante* de una variable. Para obtener más información, vea [Conversiones de tipos y seguridad de tipos](#).

Tipos string

En sentido estricto, el lenguaje C++ no tiene un tipo string integrado. `char` y `wchar_t` almacenan caracteres individuales; es necesario declarar una matriz de estos tipos para aproximarse a una cadena y agregar un valor final null (por ejemplo, `'\0'` en ASCII) al elemento de matriz después del último carácter válido (también denominado *cadena de estilo C*). En las cadenas de estilo C, era necesario escribir mucho más código o usar funciones de bibliotecas de utilidades de cadena externas. Pero en el lenguaje C++ actual, tenemos los tipos de la biblioteca estándar `std::string` (para cadenas de caracteres de tipo `char` de 8 bits) o `std::wstring` (para cadenas de caracteres de tipo `wchar_t` de 16 bits). Estos contenedores de la biblioteca estándar de C++ se pueden considerar como tipos de cadena nativos porque forman parte de las bibliotecas estándar que se incluyen en cualquier entorno de compilación de C++ compatible. Use la `#include <string>` directiva para que estos tipos estén disponibles en el programa. (Si usa MFC o ATL, la `cstring` clase también está disponible, pero no forma parte del estándar de C++). No se recomienda el uso de matrices de caracteres terminadas en null (las cadenas de estilo C mencionadas anteriormente) en C++ moderno.

Tipos definidos por el usuario

Cuando se define un objeto `class`, `struct`, `union` o `enum`, esa construcción se usa en el resto del código como si fuera un tipo fundamental. Esa construcción tiene un tamaño conocido en memoria y se aplican ciertas reglas sobre su uso durante la comprobación en tiempo de compilación y, en tiempo de ejecución, durante la vida útil del programa. Las diferencias principales entre los tipos fundamentales integrados y los tipos definidos por el usuario son las siguientes:

- El compilador no tiene conocimiento integrado de un tipo definido por el usuario. El compilador conoce el tipo la primera vez que encuentra la definición durante el proceso de compilación.
- El usuario especifica las operaciones que se pueden realizar en el tipo y cómo se puede convertir en otros tipos definiendo (mediante sobrecarga) los operadores adecuados, como los miembros de clase o las funciones que no son miembro. Para más información, consulte [Sobrecarga de funciones](#).

Tipos de puntero

Como en las primeras versiones del lenguaje C, C++ sigue permitiendo declarar una variable de un tipo de puntero mediante el declarador `*` especial (asterisco). Un tipo de puntero almacena la dirección de la ubicación en memoria donde se almacena el valor de datos real. En C++ moderno, estos tipos de puntero se conocen como *punteros sin*

procesar y se accede a ellos en el código a través de operadores especiales: `*` (asterisco) o `->` (guión con mayor que, a menudo denominado *flecha*). Esta operación de acceso a memoria se denomina *desreferenciación*. El operador que use depende de si va a desreferenciar un puntero a un escalar o un puntero a un miembro de un objeto .

Trabajar con tipos de puntero ha sido uno de los aspectos más difíciles y confusos del desarrollo de programación de C y C++. En esta sección se describen algunos hechos y prácticas para ayudar a usar punteros sin procesar si quiere. Sin embargo, en C++ moderno, ya no es necesario (o recomendado) usar punteros sin procesar para la propiedad de objetos en absoluto, debido a la evolución del [puntero inteligente](#) (que se describe más al final de esta sección). Sigue siendo útil y seguro usar punteros sin procesar para observar objetos. Sin embargo, si debe usarlos para la propiedad del objeto, debe hacerlo con precaución y teniendo en cuenta cuidadosamente cómo se crean y destruyen los objetos que les pertenecen.

Lo primero que debe saber es que una declaración de variable de puntero sin procesar solo asigna suficiente memoria para almacenar una dirección: la ubicación de memoria a la que hace referencia el puntero cuando se desreferencia. La declaración de puntero no asigna la memoria necesaria para almacenar el valor de datos. (Esa memoria también se denomina *memoria auxiliar*). Es decir, al declarar una variable de puntero sin procesar, se crea una variable de dirección de memoria, no una variable de datos real. Si desreferencia una variable de puntero antes de asegurarse de que contiene una dirección válida para un almácén de respaldo, provoca un comportamiento no definido (normalmente un error irrecuperable) en el programa. En el siguiente ejemplo se muestra este tipo de error:

C++

En el ejemplo se desreferencia un tipo de puntero que no tiene ninguna memoria asignada para almacenar los datos enteros reales ni una dirección de memoria válida asignada. El código siguiente corrige esto errores:

C++

```
// 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".
```

En el ejemplo de código corregido se utiliza la memoria local de la pila para crear la memoria auxiliar a la que `pNumber` apunta. Utilizamos un tipo fundamental para simplificar. En la práctica, los almacenes de respaldo para punteros suelen ser tipos definidos por el usuario que se asignan dinámicamente en un área de memoria denominada *montón* (o *almacén libre*) mediante una `new` expresión de palabra clave (en programación de estilo C, se usó la función anterior `malloc()` de la biblioteca en tiempo de ejecución de C). Una vez asignadas, estas variables se conocen normalmente como *objetos*, especialmente si se basan en una definición de clase. La memoria que se asigna con `new` debe eliminarse mediante la instrucción `delete` correspondiente (o, si usó la función `malloc()` para asignarlas, la función `free()` en tiempo de ejecución de C).

Sin embargo, es fácil olvidarse de eliminar un objeto asignado dinámicamente, especialmente en código complejo, lo que provoca un error de recursos denominado *pérdida de memoria*. Por este motivo, se desaconseja el uso de punteros sin procesar en C++ moderno. Casi siempre es mejor encapsular un puntero sin procesar en un **puntero inteligente**, que libera automáticamente la memoria cuando se invoca su destructor. (Es decir, cuando el código sale del ámbito del puntero inteligente). Mediante el uso de punteros inteligentes, prácticamente elimina toda una clase de errores en los programas de C++. En el ejemplo siguiente, suponga que `MyClass` es un tipo definido por el usuario que tiene un método público `DoSomeWork();`

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.
```

Para más información sobre los punteros inteligentes, consulte [Punteros inteligentes](#).

Para obtener más información sobre las conversiones de puntero, vea [Conversiones de tipos y seguridad de tipos](#).

Para obtener información sobre los punteros en general, consulte [Punteros](#).

Tipos de datos de Windows

En la programación Win32 clásica de C y C++, la mayoría de las funciones usan definiciones de tipos y macros `windef.h` (definidas en `#define`) específicas de Windows para indicar los tipos de parámetros y los valores devueltos. Estos tipos de datos de Windows son principalmente nombres especiales (alias) proporcionados a tipos integrados de C/C++. Para obtener una lista completa de estas definiciones de tipo y las definiciones de preprocesador, consulte [Tipos de datos de Windows](#). Algunas de estas definiciones de tipos, como `HRESULT` y `LCID`, son útiles y significativas. Otras, como `INT`, no tienen ningún significado especial y son solo alias para los tipos fundamentales de C++. Otros tipos de datos de Windows tienen nombres que se provienen de la época de programación de C y de los procesadores de 16 bits, y no tienen ningún propósito o significado en el hardware y sistemas operativos modernos. Hay también tipos de datos especiales asociados a la biblioteca de Windows Runtime, que se muestran como [tipos de datos base de Windows Runtime](#). En C++moderno, la guía general es preferir los tipos fundamentales de C++ a menos que el tipo de Windows comunique algún significado adicional sobre cómo se interpretará el valor.

Más información

Para obtener más información sobre el sistema de tipos de C++, consulte los siguientes artículos.

[Tipos de valor](#)

Describe los *tipos de valor* junto con problemas relacionados con su uso.

[Conversiones de tipos y seguridad de tipos](#)

Describe problemas de conversión de tipos comunes y muestra cómo evitarlos.

Consulte también

[Aquí está otra vez C++](#)

[Referencia del lenguaje C++](#)

[Biblioteca estándar de C++](#)

Ámbito (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 4 minutos

Cuando se declara un elemento de programa como una clase, función o variable, su nombre solo se puede "ver" y usar en determinadas partes del programa. El contexto en el que se ve un nombre se denomina *ámbito*. Por ejemplo, si declara una variable `x` dentro de una función, `x` solo es visible dentro de ese cuerpo de la función. Tiene *ámbito local*. Es posible que tenga otras variables con el mismo nombre en el programa; siempre que estén en distintos ámbitos, no infringen la regla de definición única y no se genera ningún error.

Para las variables no estáticas automáticas, el ámbito también determina cuándo se crean y destruyen en la memoria del programa.

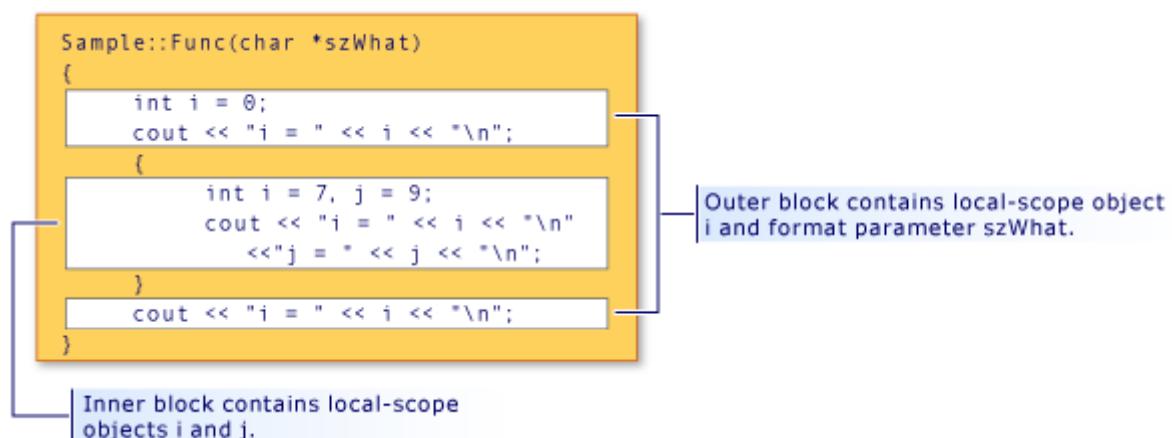
Hay seis tipos de ámbito:

- **Ámbito global** Un nombre global es uno que se declara fuera de cualquier clase, función o espacio de nombres. Sin embargo, en C++ incluso estos nombres existen con un espacio de nombres global implícito. El ámbito de los nombres globales se extiende desde el punto de declaración hasta el final del archivo en el que se declaran. En el caso de los nombres globales, la visibilidad también se rige por las reglas de [vinculación](#) que determinan si el nombre es visible en otros archivos del programa.
- **Ámbito del espacio de nombres** Un nombre que se declara dentro de un [espacio de nombres](#), fuera de cualquier clase o definición de enumeración o bloque de función, es visible desde su punto de declaración hasta el final del espacio de nombres. Un espacio de nombres se puede definir en varios bloques en distintos archivos.
- **Ámbito local** Un nombre declarado dentro de una función o lambda, incluidos los nombres de parámetro, tienen ámbito local. A menudo se conocen como "locales". Solo son visibles desde su punto de declaración hasta el final de la función o cuerpo lambda. El ámbito local es un tipo de ámbito de bloque, que se describe más adelante en este artículo.
- **Ámbito de clase** Los nombres de los miembros de clase tienen ámbito de clase, que se extiende a lo largo de la definición de clase independientemente del punto de declaración. La accesibilidad de los miembros de clase se controla aún más mediante las palabras clave `public`, `private` y `protected`. Solo se puede acceder a los miembros públicos o protegidos mediante los operadores de selección de miembros (`.` o `->`) o operadores de puntero a miembro (`.*` o `->*`).

- **Ámbito de instrucción** Los nombres declarados en una instrucción `for`, `if`, `while` o `switch` son visibles hasta el final del bloque de instrucciones.
- **Ámbito de función** Una `etiqueta` tiene ámbito de función, lo que significa que es visible a lo largo de un cuerpo de función incluso antes de su punto de declaración. El ámbito de función permite escribir instrucciones como `goto cleanup` antes de declarar la etiqueta `cleanup`.

Ocultar nombres

Puede ocultar un nombre declarándolo en un bloque delimitado. En la ilustración siguiente, `i` se declara dentro del bloque interno, ocultando de esta manera la variable asociada a `i` en el ámbito del bloque externo.



Ámbito de bloque y ocultación de nombres

El resultado del programa que se muestra en la figura es:

```
C++
i = 0
i = 7
j = 9
i = 0
```

ⓘ Nota

El argumento `szwhat` se considera en el ámbito de la función. Por consiguiente, se trata como si se hubiera declarado en el bloque exterior de la función.

Ocultar nombres de clase

Puede ocultar nombres de clase declarando una función, un objeto o una variable, o un enumerador en el mismo ámbito. Sin embargo, aún se puede tener acceso al nombre de clase cuando va precedido por la palabra clave `class`.

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
```

ⓘ Nota

En cualquier lugar donde se llame al nombre de clase (`Account`), se debe usar la palabra clave `class` para diferenciarla de la variable `Account` del ámbito global. Esta regla no se aplica cuando el nombre de clase aparece a la izquierda del operador de resolución de ámbito (`::`). Los nombres del lado izquierdo del operador de resolución de ámbito siempre se consideran nombres de clase.

En el ejemplo siguiente se muestra cómo declarar un puntero a un objeto de tipo `Account` mediante la palabra clave `class`:

C++

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

`Account` en el inicializador (entre paréntesis) de la instrucción anterior tiene ámbito de archivo; es de tipo `double`.

ⓘ Nota

La reutilización de los nombres de identificador, tal y como se muestra en este ejemplo, se considera mal estilo de programación.

Para obtener información sobre la declaración y la inicialización de los objetos de clase, vea [Clases, estructuras y uniones](#). Para obtener información sobre cómo utilizar los operadores de almacenamiento libre `new` y `delete`, vea [Operadores new y delete](#).

Ocultar nombres con ámbito global

Puede ocultar nombres con ámbito global declarando explícitamente el mismo nombre en ámbito de bloque. Sin embargo, se puede obtener acceso a los nombres de ámbito global usando el operador de resolución de ámbito (`::`).

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
```

Consulte también

[Conceptos básicos](#)

Archivos de encabezado (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 4 minutos

Los nombres de elementos de programa, como variables, funciones, clases, etc., deben declararse antes de poder usarse. Por ejemplo, no puede solo escribir `x = 42` sin declarar primero "x".

C++

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

La declaración indica al compilador si el elemento es un `int`, un `double`, una función, un `class` o alguna otra cosa. Además, cada nombre debe declararse (directa o indirectamente) en cada archivo .cpp en el que se use. Al compilar un programa, cada archivo .cpp se compila de forma independiente en una unidad de compilación. El compilador no tiene conocimiento de los nombres que se declaran en otras unidades de compilación. Esto significa que si define una clase o función o variable global, debe proporcionar una declaración de ese elemento en cada archivo .cpp adicional que lo use. Cada declaración de ese elemento debe ser exactamente idéntica en todos los archivos. Una ligera incoherencia provocará errores, o un comportamiento no deseado, cuando el enlazador intente combinar todas las unidades de compilación en un único programa.

Para minimizar el potencial de errores, C++ ha adoptado la convención de usar *archivos de encabezado* para contener declaraciones. Usted realiza las declaraciones en un archivo de encabezado y, a continuación, se usa la directiva `#include` en cada archivo .cpp u otro archivo de encabezado que requiera esa declaración. La directiva `#include` inserta una copia del archivo de encabezado directamente en el archivo .cpp antes de la compilación.

ⓘ Nota

En Visual Studio 2019, la característica de *módulos* de C++20 se presenta como una mejora y reemplazo posible de los archivos de encabezado. Para obtener más información, consulte [Información general de los módulos en C++](#).

Ejemplo

El ejemplo siguiente muestra una manera común de declarar una clase y, a continuación, usarla en un archivo de código fuente diferente. Comenzaremos con el archivo de encabezado, `my_class.h`. Contiene una definición de clase, pero tenga en cuenta que la definición está incompleta; la función miembro `do_something` no está definida:

C++

```
// my_class.h
namespace N
{
    class my_class
    {
        public:
            void do_something();
    };
}
```

A continuación, cree un archivo de implementación (normalmente con una extensión .cpp o similar). Llame a este archivo `my_class.cpp` y proporcionaremos una definición para la declaración de miembro. Agregamos una directiva `#include` para el archivo "my_class.h" para que la declaración de `my_class` se inserte en este punto en el archivo .cpp e incluiremos `<iostream>` para extraer la declaración de `std::cout`. Tenga en cuenta que las comillas se usan para los archivos de encabezado en el mismo directorio que el archivo de origen y los corchetes angulares se usan para encabezados de biblioteca estándar. Además, muchos encabezados de biblioteca estándar no tienen .h ni ninguna otra extensión de archivo.

En el archivo de implementación, opcionalmente podemos usar una instrucción `using` para evitar tener que calificar todas las menciones de "my_class" o "cout" con "N::" o "std::". No coloque instrucciones `using` en los archivos de encabezado.

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;
}
```

Ahora podemos usar `my_class` en otro archivo .cpp. Usamos `#include` en el archivo de encabezado para que el compilador extraiga la declaración. Todo el compilador debe saber que `my_class` es una clase que tiene una función miembro pública denominada `do_something()`.

C++

```
// my_program.cpp
#include "my_class.h"

using namespace N;

int main()
{
    my_class mc;
    mc.do_something();
    return 0;
}
```

Después de que el compilador termine de compilar cada archivo .cpp en archivos .obj, pasa los archivos .obj al enlazador. Cuando el enlazador combina los archivos de objeto, encuentra exactamente una definición para `my_class`; está en el archivo .obj generado para `my_class.cpp` y la compilación se realiza correctamente.

Incluir restricciones

Normalmente, los archivos de encabezado tienen una directiva *incluir restricciones* o `#pragma once` para asegurarse de que no se insertan varias veces en un único archivo .cpp.

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 */
```

Qué colocar en un archivo de encabezado

Dado que un archivo de encabezado podría incluirse en varios archivos, no puede contener definiciones que puedan generar varias definiciones del mismo nombre. No se permite lo siguiente, o se considera una práctica muy desaconsejable:

- definiciones de tipo integradas en el espacio de nombres o el ámbito global
- definiciones de funciones no insertadas
- definiciones de variables no constantes
- definiciones de agregado
- espacios de nombres sin nombre
- Directivas using

El uso de la directiva `using` no provocará necesariamente un error, pero puede causar un problema porque lleva el espacio de nombres al ámbito en cada archivo .cpp que incluya directa o indirectamente ese encabezado.

Archivo de encabezado de ejemplo

En el ejemplo siguiente se muestran los distintos tipos de declaraciones y definiciones que se permiten en un archivo de encabezado:

```
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;
}
```

Unidades de traducción y vinculación

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

En un programa de C++, un *símbolo*, por ejemplo, una variable o un nombre de función, se puede declarar cualquier número de veces dentro de su ámbito. Sin embargo, solo puede definirse una vez. Esta regla es la "Regla de definición única" (ODR). Una *declaración* introduce (o reintroduce) un nombre en el programa junto con suficiente información para asociar posteriormente el nombre a una definición. Una *definición* introduce un nombre y proporciona toda la información necesaria para crearlo. Si el nombre representa una variable, una definición crea explícitamente el almacenamiento y lo inicializa. Una *definición de función* consta de la firma más el cuerpo de la función. Una definición de clase consta del nombre de clase seguido de un bloque que enumera todos los miembros de la clase. (Los cuerpos de las funciones miembro se pueden definir opcionalmente por separado en otro archivo).

En el siguiente ejemplo se muestran algunas declaraciones:

C++

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

En el siguiente ejemplo se muestran algunas definiciones:

C++

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

Un programa consta de una o más *unidades de traducción*. Una unidad de traducción consta de un archivo de implementación y de todos los encabezados que incluye directa o indirectamente. Normalmente, los archivos de implementación tienen una extensión de archivo de `.cpp` o `.cxx`. Normalmente, los archivos de encabezado tienen una extensión de `.h` o `.hpp`. Cada unidad de traducción es compilada independientemente por el compilador. Una vez completada la compilación, el enlazador combina las unidades de traducción compiladas en un único *programa*. Las infracciones de la regla

de ODR suelen aparecer como errores del vinculador. Los errores del enlazador se producen cuando el mismo nombre se define en más de una unidad de traducción.

En general, la mejor manera de hacer que una variable sea visible en varios archivos es declararla en un archivo de encabezado. A continuación, agregue una directiva `#include` en cada archivo `.cpp` que requiera la declaración. Al agregar *protección de inclusión* alrededor del contenido del encabezado, asegúrese de que los nombres que declara un encabezado solo se declaran una vez para cada unidad de traducción. Defina el nombre en un solo archivo de implementación.

En C++20, los [módulos](#) se presentan como una alternativa mejorada para los archivos de encabezado.

En algunos casos, puede ser necesario declarar una variable global o una clase en un archivo `.cpp`. En esos casos, se necesita una manera de indicar al compilador y al enlazador qué tipo de *vinculación* tiene el nombre. El tipo de vinculación especifica si el nombre del objeto solo está visible en un archivo o en todos los archivos. El concepto de vinculación solo se aplica a los nombres globales. El concepto de vinculación no se aplica a los nombres que se declaran dentro de un ámbito. Un ámbito se especifica mediante un conjunto de llaves que lo encierran, como en las definiciones de función o clase.

Vinculación externa frente a interna

Una *función libre* es una función que se define en el ámbito global o del espacio de nombres. De forma predeterminada, las variables globales no-`const` y las funciones libres tienen *vinculación externa*; son visibles desde cualquier unidad de traducción del programa. Ningún otro objeto global puede tener ese nombre. Un símbolo con *vinculación interna* o *ninguna vinculación* solo está visible dentro de la unidad de traducción en la que se declara. Cuando un nombre tiene vinculación interna, el mismo nombre puede existir en otra unidad de traducción. Las variables declaradas dentro de las definiciones de clase o los cuerpos de función no tienen vinculación.

Puede forzar que un nombre global tenga vinculación interna declarándolo explícitamente como `static`. Esta palabra clave limita su visibilidad a la misma unidad de traducción en la que se declara. En este contexto, `static` significa algo diferente que cuando se aplica a variables locales.

Los objetos siguientes tienen vinculación interna de forma predeterminada:

- Objetos `const`.
- Objetos `constexpr`.

- Objetos `typedef`.
- Objetos `static` en el ámbito del espacio de nombres

Para proporcionar una vinculación externa de objeto `const`, declárela como `extern` y asígnele un valor:

C++

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

Para obtener más información, vea [extern](#).

Vea también

[Conceptos básicos](#)

Función `main` y argumentos de la línea de comandos

Artículo • 03/03/2023 • Tiempo de lectura: 9 minutos

Todos los programas de C++ deben tener una función `main`. Si se intenta compilar un programa de C++ sin una función `main`, el compilador genera un error. (Las bibliotecas `static` y las bibliotecas de vínculos dinámicos carecen de función `main`). La función `main` es donde el código fuente empieza a ejecutarse, pero antes de que un programa entre en la función `main`, todos los miembros de clase `static` sin inicializadores explícitos se establecen en cero. En Microsoft C++, los objetos `static` globales también se inicializan antes de entrar en `main`. A la función `main` se le aplican varias restricciones que no se aplican a otras funciones de C++. La función `main`:

- No se puede sobrecargar (vea [Sobrecarga de funciones](#)).
- No se puede declarar como `inline`.
- No se puede declarar como `static`.
- Su dirección no se puede tomar.
- No se puede llamar desde un programa de un usuario.

Signatura de la función `main`

La función `main` no tiene una declaración porque está integrada en el lenguaje. Si la tuviera, la sintaxis de declaración de `main` tendría este aspecto:

C++

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

Si no se especifica ningún valor devuelto en `main`, el compilador proporciona un valor devuelto de cero.

Argumentos de línea de comandos estándar

Los argumentos para `main` permiten realizar un práctico análisis de línea de comandos de los argumentos. El lenguaje define los tipos `argc` y `argv`. Los nombres `argc` y `argv` son tradicionales, pero se les puede asignar el nombre que se quiera.

Las definiciones de los argumentos son las siguientes:

`argc`

Un entero que contiene el número de argumentos que aparecen detrás de `argv`. El parámetro `argc` es siempre mayor o igual que 1.

`argv`

Una matriz de cadenas terminadas en null que representan los argumentos de la línea de comandos especificados por el usuario del programa. Por convención, `argv[0]` es el comando con el que se invoca el programa. `argv[1]` es el primer argumento de la línea de comandos. El último argumento de la línea de comandos es `argv[argc - 1]`, y `argv[argc]` siempre es NULL.

Para obtener información sobre cómo suprimir el procesamiento de la línea de comandos, vea [Personalización del procesamiento de la línea de comandos de C++](#).

ⓘ Nota

Por convención, `argv[0]` es el nombre de archivo del programa. Sin embargo, en Windows es posible generar un proceso mediante `CreateProcess`. Si se usa tanto el primer argumento como el segundo (`LpApplicationName` y `LpCommandLine`), `argv[0]` no puede ser el nombre del archivo ejecutable. Puede usar `GetModuleFileName` para recuperar el nombre del archivo ejecutable y su ruta de acceso completa.

Extensiones específicas de Microsoft

En las siguientes secciones se describe el comportamiento específico de Microsoft.

La función `wmain` y la macro `_tmain`

Si diseña el código fuente para que use characters anchos Unicode, puede usar el punto de entrada `wmain` específico de Microsoft, que es la versión de characters ancho de `main`. Esta es la sintaxis de declaración de facto de `wmain`:

C++

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

También puede usar la macro `_tmain` específica de Microsoft, que es una macro de preprocesador definida en `tchar.h`. `_tmain` se resuelve en `main` a menos que se defina `_UNICODE`. En ese caso, `_tmain` se resuelve en `wmain`. La macro `_tmain` y otras macros que comienzan por `_t` son útiles en códigos que deben compilar versiones independientes de los juegos de caracteres anchos y estrechos. Para obtener más información, vea [Uso de asignaciones de texto genérico](#).

void devuelto de main

Como extensión de Microsoft, las funciones `main` y `wmain` se pueden declarar para que devuelvan `void` (ningún valor devuelto). Esta extensión también está disponible en otros compiladores, pero su uso no se recomienda. Está disponible para simetría cuando `main` no devuelve un valor.

Si declara que `main` o `wmain` devuelvan `void`, no se puede devolver un código exit al proceso primario o al sistema operativo mediante una instrucción `return`. Para devolver un código exit cuando `main` o `wmain` se declaran como `void`, debe utilizar la función `exit`.

El argumento de la línea de comandos `envp`

Las signaturas `main` o `wmain` permiten una extensión específica de Microsoft opcional para acceder a variables de entorno. Esta extensión también es común en otros compiladores de sistemas Windows y UNIX. El nombre del parámetro de entorno `envp` es tradicional, pero puede llamarlo como quiera. Estas son las declaraciones de facto de las listas de argumentos que incluyen el parámetro de entorno:

C++

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

`envp`

El parámetro `envp` opcional es una matriz de cadenas que representan las variables establecidas en el entorno de usuario. Esta matriz finaliza mediante una entrada NULL. Se puede declarar como una matriz de punteros a `char` (`char *envp[]`) o como un puntero a punteros a `char` (`char **envp`). Si su programa usa `wmain` en lugar de `main`, utilice el tipo de datos `wchar_t` en lugar de `char`.

El bloque de entorno que se pasa a `main` y a `wmain` es una copia "inmóvil" del entorno actual. Si, más adelante, cambia el entorno llamando a `putenv` o a `_wputenv`, el entorno actual (devuelto por `getenv` o `_wgetenv`, o por las variables `_environ` o `_wenviron`) cambiará, pero no el bloque al que `envp` apunta. Para obtener más información sobre cómo suprimir el procesamiento del entorno, vea [Personalización del procesamiento de la línea de comandos de C++](#). El argumento `envp` es compatible con el estándar C89, pero no con los estándares de C++.

Argumentos de ejemplo en `main`

En el ejemplo siguiente se muestra cómo usar los argumentos `argc`, `argv` y `envp` en `main`:

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) && _stricmp( 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";
    }
}
```

Analizar los argumentos de la línea de comandos de C++

Las reglas de análisis de la línea de comandos usadas por el código de Microsoft C/C++ son específicas de Microsoft. El código de inicio del entorno de ejecución utiliza estas

reglas al interpretar los argumentos proporcionados en la línea de comandos del sistema operativo:

- Los argumentos van delimitados por espacio en blanco, que puede ser un carácter de espacio o una tabulación.
- El primer argumento (`argv[0]`) se trata de forma especial. Representa el nombre del programa. Dado que debe ser un nombre de ruta válido, se permiten partes entre comillas dobles ("). Las comillas dobles no se incluyen en la salida `argv[0]`. Las partes delimitadas por comillas dobles impiden la interpretación de un carácter de espacio o tabulación como final del argumento. No se aplican las últimas reglas de esta lista.
- Una cadena delimitada por comillas dobles se interpreta como un solo argumento, que puede contener caracteres de espacios en blanco. Se puede incrustar una cadena entre comillas dentro de un argumento. El símbolo de inserción (^) no se reconoce como carácter de escape ni como delimitador. Dentro de una cadena entrecomillada, un par de comillas dobles se interpreta como una sola marca de comilla doble con escape. Si la línea de comandos finaliza antes de que se encuentre una comilla doble de cierre, todos los caracteres leídos hasta ahora se muestran como el último argumento.
- Un signo de comillas dobles precedido por una barra diagonal inversa (\") se interpreta como signo de comillas dobles literal ("").
- Las barras diagonales inversas se interpretan literalmente, a menos que precedan inmediatamente a unas comillas dobles.
- Si un número par de barras diagonales inversas va seguido de un signo de comillas dobles, se coloca una barra diagonal inversa (\) en la matriz `argv` por cada par de barras diagonales inversas (\) y el signo de comillas dobles (") se interpreta como delimitador de cadena.
- Si un número impar de barras diagonales inversas va seguido de una comilla doble, se coloca una barra diagonal inversa (\) en la matriz `argv` por cada par de barras diagonales inversas (\). La barra diagonal restante interpreta la comilla doble como una secuencia de escape, lo que hace que se coloque una marca de comilla doble literal (") en `argv`.

Ejemplo de análisis de argumentos de la línea de comandos

En el programa siguiente se muestra cómo se pasan los argumentos de la línea de comandos:

```
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";  
}
```

Resultados del análisis de las líneas de comandos

En la tabla siguiente se muestra una entrada de ejemplo y la salida esperada para mostrar las reglas de la lista anterior.

| Entrada de la línea de comandos | 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 |
| a\\\\\"b c" d e | a\\b c | d | e |
| a"b"" c d | ab" | c d | |

Expansión de caracteres comodín

Opcionalmente, el compilador de Microsoft permite usar caracteres *comodín*, el signo de interrogación (?) y asteriscos (*) para especificar argumentos de nombre de archivo y ruta de acceso en la línea de comandos.

Los argumentos de la línea de comandos se controlan mediante una rutina interna del código de inicio del entorno de ejecución que, de forma predeterminada, no expande los caracteres comodín en cadenas independientes en la matriz de cadenas `argv`. Para permitir la expansión de caracteres comodín, puede incluir el archivo `setargv.obj` (archivo `wsetargv.obj` si es `wmain`) en las opciones del compilador `/link` o en la línea de comandos `LINK`.

Para obtener más información sobre las opciones del vinculador de inicio en tiempo de ejecución, consulte [Opciones de vínculo](#).

Personalización del procesamiento de línea de comandos de C++

Si el programa no acepta argumentos de línea de comandos, puede suprimir la rutina de procesamiento de la línea de comandos para guardar una pequeña cantidad de espacio. Para suprimir su uso, incluya el archivo `noarg.obj` (para `main` y `wmain`) en sus opciones del compilador `/link` o su línea de comandos `LINK`.

Del mismo modo, si nunca tiene acceso a la tabla de entorno a través del argumento `envp`, puede suprimir la rutina de procesamiento del entorno interna. Para suprimir su uso, incluya el archivo `noenv.obj` (para `main` y `wmain`) en sus opciones del compilador `/link` o su línea de comandos `LINK`.

El programa puede realizar llamadas a la familia de rutinas `spawn` o `exec` de la biblioteca en tiempo de ejecución de C. Si lo hace, no debe suprimir la rutina de procesamiento de entorno, puesto que se utiliza para pasar un entorno del proceso primario al proceso secundario.

Consulte también

[Conceptos básicos](#)

Finalización del programa C++

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

En C++, puede salir de un programa de estas maneras:

- Llame a la función `exit`.
- Llame a la función `abort`.
- Ejecute una instrucción `return` desde `main`.

Función `exit`

La función `exit`, declarada en `<stdlib.h>`, finaliza un programa de C++. El valor proporcionado como argumento a `exit` se devuelve al sistema operativo como código de retorno o código de salida del programa. Por convención, un código de retorno de cero significa que el programa se completó correctamente. Puede utilizar las constantes `EXIT_FAILURE` y `EXIT_SUCCESS`, también definidas en `<stdlib.h>`, para indicar si el programa se ha completado correctamente o ha generado errores.

Función `abort`

La función `abort`, declarada también en el archivo de inclusión estándar `<stdlib.h>`, finaliza un programa de C++. La diferencia entre `exit` y `abort` es que `exit` permite que tenga lugar el procesamiento de finalización en tiempo de ejecución de C++ (se llamará a los destructores de objetos globales). `abort` finaliza el programa inmediatamente. La función `abort` omite el proceso de destrucción normal de los objetos estáticos globales inicializados. También omite cualquier procesamiento especial especificado mediante la función `atexit`.

Específico de Microsoft: por motivos de compatibilidad de Windows, la implementación de Microsoft de `abort` puede permitir que el código de terminación DLL se ejecute en determinadas circunstancias. Para más información, consulte [abort](#).

Función `atexit`

Use la función `atexit` para especificar las acciones que se ejecutan antes de que finalice el programa. Los objetos estáticos no globales inicializados antes de la llamada a `atexit` se destruyen antes de la ejecución de la función de procesamiento de salida.

Instrucción `return` en `main`

La instrucción `return` le permite especificar un valor devuelto desde `main`. Una instrucción `return` en `main` primero actúa como cualquier otra instrucción `return`. Las variables automáticas se destruyen. A continuación, `main` invoca `exit` con el valor devuelto como parámetro. Considere el ejemplo siguiente:

C++

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

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

Las instrucciones `exit` y `return` del ejemplo anterior tienen un comportamiento similar. Ambos finalizan el programa y devuelven un valor de 3 al sistema operativo. La diferencia es que `exit` no destruye la variable automática `s`, mientras que la instrucción `return` sí.

Por lo general, C++ requiere que las funciones que tienen tipos de valor devuelto distintos de `void` devuelvan un valor. La función `main` es una excepción; puede terminar sin una instrucción `return`. En ese caso, devuelve un valor específico de la implementación al proceso de invocación. (De forma predeterminada, MSVC devuelve 0).

Destrucción de subprocessos y objetos estáticos

Cuando se llama `exit` directamente (o cuando se llama después de una instrucción `return` de `main`), se destruyen los objetos de subprocesso asociados al subprocesso actual. A continuación, los objetos estáticos se destruyen en el orden inverso de su inicialización (después de las llamadas a funciones especificadas en `atexit`, si existen). En el ejemplo siguiente se muestra cómo funcionan esa inicialización y limpieza.

Ejemplo

En el ejemplo siguiente, los objetos estáticos `sd1` y `sd2` se crean e inicializan antes de la entrada a `main`. Después de que finalice este programa mediante la instrucción `return`, primero se destruye `sd2` y después `sd1`. El destructor para la clase de `ShowData` cierra los archivos asociados a estos objetos estáticos.

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" );
}
```

Otra forma de escribir este código es declarar los objetos `ShowData` con ámbito de bloque, lo que permite destruirlos cuando salen del ámbito:

C++

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

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

Consulte también

[Función main y argumentos de la línea de comandos](#)

Lvalues y rvalues (C++)

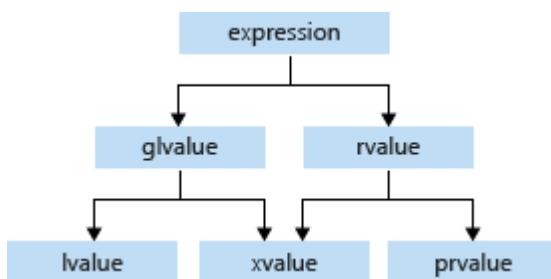
Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Cada expresión de C++ tiene un tipo y pertenece a una *categoría de valor*. Las categorías de valor son la base de las reglas que los compiladores deben seguir al crear, copiar y mover objetos temporales durante la evaluación de expresiones.

El estándar de C++17 define las categorías de valores de expresión de la siguiente manera:

- Un *glvalue* es una expresión cuya evaluación determina la identidad de un objeto, un campo de bits o una función.
- Un *valor prvalue* es una expresión cuya evaluación inicializa un objeto o un campo de bits, o calcula el valor del operando de un operador, tal como se especifica en el contexto en el que aparece.
- Un *valor xvalue* es un valor glvalue que denota un objeto o campo de bits cuyos recursos se pueden reutilizar (normalmente porque está cerca del final de su vigencia). Ejemplo: Ciertos tipos de expresiones que implican referencias rvalue (8.3.2) producen valores xvalue, como una llamada a una función cuyo tipo de valor devuelto es una referencia rvalue o una conversión a un tipo de referencia rvalue.
- Un *valor lvalue* es un valor glvalue que no es un valor xvalue.
- Un *valor rvalue* es un valor prvalue o un valor xvalue.

En el diagrama siguiente se ilustran las relaciones entre las categorías:



Un valor lvalue tiene una dirección a la que puede acceder el programa. Algunos ejemplos de expresiones lvalue incluyen nombres de variable, como variables `const`, elementos de matriz, llamadas de función que devuelven una referencia lvalue, campos de bits, uniones y miembros de clase.

Una expresión prvalue no tiene ninguna dirección a la que el programa pueda acceder. Algunos ejemplos de expresiones prvalue incluyen literales, llamadas de función que devuelven un tipo que no es de referencia y objetos temporales creados durante la evaluación de expresiones, pero accesibles solo por el compilador.

Una expresión `xvalue` tiene una dirección que ya no es accesible por el programa, pero que se puede usar para inicializar una referencia `rvalue`, que proporciona acceso a la expresión. Entre los ejemplos se incluyen llamadas de función que devuelven una referencia `rvalue` y las expresiones de subíndice de matriz, miembro y puntero a miembro en las que la matriz o el objeto es una referencia `rvalue`.

Ejemplo

En el ejemplo siguiente se muestran varios usos correctos e incorrectos de valores L y valores R:

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
}
```

ⓘ Nota

En los ejemplos de este tema se muestra el uso correcto e incorrecto cuando los operadores no están sobrecargados. Si sobrecarga operadores, puede convertir una expresión tal como `j * 4` en un valor L.

Los términos *lvalue* y *rvalue* suelen utilizarse para referirse a referencias a objetos. Para obtener más información sobre las referencias, vea [Declarador de referencia lvalue: &](#) y

Declarador de referencias rvalue: `&&`.

Consulte también

[Conceptos básicos](#)

[Declarador de referencias lvalue: `&`](#)

[Declarador de referencias rvalue: `&&`](#)

Objetos temporales

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Un objeto temporal es un objeto sin nombre creado por el compilador para almacenar un valor temporal.

Comentarios

En algunos casos, es necesario que el compilador cree objetos temporales. Estos objetos temporales se pueden crear por las razones siguientes:

- Para inicializar una referencia `const` con un inicializador de un tipo diferente del tipo subyacente de la referencia que se va a inicializar.
- Para almacenar el valor devuelto de una función que devuelve un tipo definido por el usuario (UDT). Estos objetos temporales solo se crean si el programa no copia el valor devuelto en un objeto. Por ejemplo:

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.
```

Como el valor devuelto no se copia en otro objeto, se crea un objeto temporal. Un caso más común de creación de objetos temporales es durante la evaluación de una expresión en la que deben llamarse a funciones de operador sobrecargadas. Estas funciones de operador sobrecargadas devuelven un tipo definido por el usuario que normalmente no se copia a otro objeto.

Considere la expresión `ComplexResult = Complex1 + Complex2 + Complex3`. La expresión `Complex1 + Complex2` se evalúa y el resultado se almacena en un objeto temporal. A continuación, se evalúa la expresión `+ Complex3 temporary` y el resultado se copia en `ComplexResult` (suponiendo que el operador de asignación no esté sobrecargado).

- Para almacenar el resultado de una conversión en un tipo definido por el usuario. Cuando un objeto de un tipo determinado se convierte explícitamente en un tipo definido por el usuario, este nuevo objeto se crea como un objeto temporal.

Los objetos temporales tienen una duración que viene definida por su punto de creación y por el punto en el que se destruyen. Cualquier expresión que cree más de un objeto temporal los acabará destruyendo en el orden inverso en que se crearon.

Cuando se produce la destrucción de un objeto temporal, depende de cómo se use:

- Objetos temporales usados para inicializar referencias `const`:
Si un inicializador no es un valor L del mismo tipo que la referencia que se va a inicializar, se crea un objeto temporal del tipo de objeto subyacente. Se inicializa mediante la expresión de inicialización. Este objeto temporal se destruye inmediatamente después de que se destruya el objeto de referencia al que está enlazado. Como esta destrucción puede ocurrir mucho después de la expresión que creó el objeto temporal, a veces se conoce como *extensión de duración*.
- Los objetos temporales creados como efecto de la evaluación de expresiones:
Todos los objetos temporales que no encajen en la primera categoría y que se creen como resultado de la evaluación de la expresión se destruyen al final de la instrucción de expresión (es decir, en el punto y coma) o al final de las expresiones de control en el caso de las instrucciones `for`, `if`, `while`, `do` y `switch`.

Consulte también

Blog de Herb Sutter en [References, simply ↗](#)

Alineación

Artículo • 03/03/2023 • Tiempo de lectura: 4 minutos

Una de las características de bajo nivel de C++ es la capacidad para especificar la alineación precisa de los objetos en la memoria para sacar el máximo partido de una arquitectura de hardware específica. De forma predeterminada, el compilador alinea los miembros de clase y estructura en su valor de tamaño: `bool` y `char` en límites de 1 byte, `short` en límites de 2 bytes, `int`, `long` y `float` en límites de 4 bytes y `long long`, `double` y `long double` en límites de 8 bytes.

En la mayoría de los escenarios no tendrá que preocuparse jamás por la alineación, puesto que la alineación predeterminada ya es óptima. No obstante, en algunos casos se pueden conseguir importantes mejoras de rendimiento o ahorros de memoria al especificar una alineación personalizada para sus estructuras de datos. Antes de Visual Studio 2015 podía usar las palabras clave `_alignof` y `_declspec(align)` específicas de Microsoft para especificar una alineación mayor que el valor predeterminado. Desde Visual Studio 2015 se deben usar las palabras clave estándar de C++11 `alignof` y `alignas` para aumentar al máximo la portabilidad de su código. Las nuevas palabras clave se comportan de la misma manera que las extensiones específicas de Microsoft. La documentación de esas extensiones también se aplica a las nuevas palabras clave. Para obtener más información, consulte [Operador alignof](#) y [align](#). El estándar de C++ no especifica el comportamiento de empaquetado para alinear en límites inferiores al valor predeterminado del compilador para la plataforma de destino, por lo que seguirá teniendo que usar el `#pragma pack` de Microsoft en ese caso.

Use la [clase aligned_storage](#) para la asignación de memoria de estructuras de datos con alineaciones personalizadas. La [clase aligned_union](#) consiste en especificar la alineación de uniones con constructores o destructores no triviales.

Alineación y direcciones de memoria

La alineación es una propiedad de una dirección de memoria, expresada como el módulo de la dirección numérica a una potencia de 2. Por ejemplo, la dirección 0x0001103F módulo 4 es 3. Esa dirección está alineada con $4n+3$, donde 4 indica la potencia de 2 elegida. La alineación de una dirección depende de la potencia de 2 elegida. El mismo módulo de dirección 8 es 7. Se dice que una dirección está alineada con X si su alineación es $Xn+0$.

Las CPU ejecutan instrucciones que funcionan con datos almacenados en memoria. Los datos se identifican mediante sus direcciones en memoria. Un solo dato también tiene

un tamaño. Un dato está *alineado naturalmente* si su dirección está alineada a su tamaño. De lo contrario, se llama *desalineado*. Por ejemplo, un dato de punto flotante de 8 bytes se encuentra alineado naturalmente si la dirección que se utiliza para identificarlo está alineada a 8.

Control del compilador de la alineación de datos

Los compiladores intentan realizar asignaciones de datos de forma que impidan la desalineación de datos.

Para los tipos de datos simples, el compilador asigna direcciones que son múltiplos del tamaño en bytes del tipo de datos. Por ejemplo, el compilador asigna direcciones a variables de tipo `long` que son múltiplos de 4, estableciendo los 2 bits inferiores de la dirección en cero.

El compilador también incluye estructuras de una manera que alinea de forma natural cada elemento de la estructura. Considere la estructura `struct x_` en el siguiente ejemplo de código:

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

El compilador rellena esta estructura para aplicar la alineación de forma natural.

En el siguiente ejemplo de código se muestra cómo el compilador coloca la estructura rellenada en memoria:

```
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];
```

Ambas declaraciones devuelven `sizeof(struct x_)` como 12 bytes.

La segunda declaración incluye dos elementos de relleno:

1. `char _pad0[3]` para alinear el miembro `int b` en un límite de 4 bytes.
2. `char _pad1[1]` para alinear los elementos de matriz de la estructura `struct _x bar[3];` en un límite de cuatro bytes.

El relleno alinea los elementos de `bar[3]` de tal forma que permite el acceso natural.

En el código de ejemplo siguiente se muestra el diseño de matriz `bar[3]`:

```
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 y alignas

El especificador de tipo `alignas` es una manera portátil y estándar de C++ de especificar una alineación personalizada de variables y tipos definidos por el usuario. El operador `alignof` también es una forma estándar y portátil de obtener la alineación de una variable o un tipo especificado.

Ejemplo

Puede usar `en una clase, struct o union, o en miembros individuales. Cuando varios especificadores de se encuentran, el compilador elige el más estricto (el que tiene el valor más grande).`

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
}
```

Consulte también

[Alineación de estructuras de datos ↗](#)

Tipos triviales, de diseño estándar, POD y literales

Artículo • 03/03/2023 • Tiempo de lectura: 6 minutos

El término *diseño* hace referencia a cómo se organizan en la memoria los miembros de un objeto de tipo de unión, estructura o clase. En algunos casos, el diseño está bien definido por la especificación del lenguaje. Pero cuando una clase o estructura contienen determinadas características de lenguaje C++ como clases base virtuales, funciones virtuales o miembros con distintos controles de acceso, el compilador es libre de elegir un diseño. Ese diseño puede variar dependiendo de las optimizaciones que se realizan y puede que, en muchos casos, el objeto ni siquiera ocupe un área contigua de memoria. Por ejemplo, si una clase tiene funciones virtuales, es posible que todas las instancias de esa clase compartan una única tabla de funciones virtuales. Estos tipos resultan muy útiles, pero también tienen limitaciones. Debido a que el diseño no está definido, no se pueden pasar a programas escritos en otros lenguajes, como C, y debido a que pueden no ser contiguos, no se pueden copiar con confianza con funciones rápidas de bajo nivel, como `memcpy`, ni serializarse a través de una red.

Para permitir a los compiladores, así como a los metaprogramas y programas de C++, razonar sobre la idoneidad de un tipo dado para las operaciones que dependen de un diseño de memoria específica, C++14 presentó tres categorías de clases y estructuras simples: *trivial*, *diseño estándar* y *POD* (Plain Old Data). La biblioteca estándar tiene las plantillas de función `is_trivial<T>`, `is_standard_layout<T>` y `is_pod<T>` que determinan si un tipo determinado pertenece a una categoría determinada.

Tipos triviales

Cuando una clase o estructura de C++ tiene funciones miembro especiales proporcionadas por el compilador o establecidas explícitamente como valor predeterminado, se trata de un tipo trivial. Ocupa un área de memoria contigua. Puede tener miembros con distintos especificadores de acceso. En C++, el compilador es libre de elegir cómo ordenar los miembros en esta situación. Por lo tanto, puede usar la función `memcpy` en dichos objetos pero no consumirlos con confianza desde un programa de C. Un tipo T trivial puede copiarse en una matriz de tipos `char` o `unsigned char` y copiarse de nuevo de forma segura en una variable T. Tenga en cuenta que, debido a los requisitos de alineación, podría haber bytes de relleno entre miembros del tipo.

Los tipos triviales tienen un constructor predeterminado trivial, un constructor de copia trivial, un operador de asignación de copia trivial y un destructor trivial. En cada caso, *trivial* significa que el operador, constructor o destructor no están proporcionados por el usuario y pertenecen a una clase que

- no tiene funciones virtuales ni clases base virtuales,
- ni tiene clases base con un constructor, operador o destructor de tipo no trivial correspondiente
- ni miembros de datos del tipo de clase con un constructor, operador o destructor de tipo no trivial correspondiente

En los siguientes ejemplos se muestran tipos triviales. En Trivial2, la presencia del constructor `Trivial2(int a, int b)` requiere que se proporcione un constructor predeterminado. Para que el tipo se considere trivial, debe establecerse explícitamente ese constructor como predeterminado.

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
};
```

Tipos de diseño estándar

Cuando una clase o estructura contienen determinadas características del lenguaje C++, como funciones virtuales que no se encuentran en el lenguaje C, y todos los miembros tienen el mismo control de acceso, se trata de un tipo de diseño estándar. Es capaz de usar la función `memcpy` y el diseño está lo suficientemente definido como para que los programas de C puedan consumirlo. Los tipos de diseño estándar pueden tener funciones miembro especiales definidas por el usuario. Además, los tipos de diseño estándar tienen estas características:

- no tienen funciones virtuales ni clases base virtuales
- todos los miembros de datos no estáticos tienen el mismo control de acceso
- todos los miembros no estáticos del tipo de clase son de diseño estándar
- las clases base son de diseño estándar
- no tienen clases base del mismo tipo como primer miembro de datos no estáticos.
- cumplen alguna de estas condiciones:
 - no tienen ningún miembro de datos no estáticos en la clase más derivada y no tienen más de una clase base con miembros de datos no estáticos, o
 - no tienen clases base con miembros de datos no estáticos

El código siguiente muestra un ejemplo de un tipo de diseño estándar:

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
};
```

Quizás se pueden ilustrar mejor los dos últimos requisitos con el código. En el ejemplo siguiente, aunque `Base` es un diseño estándar, `Derived` no es un diseño estándar porque tanto él (la clase más derivada) como `Base` tienen miembros de datos no estáticos:

C++

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

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

En este ejemplo, `Derived` es un diseño estándar porque `Base` no tiene ningún miembro de datos no estáticos:

```
C++  
  
struct Base  
{  
    void Foo() {}  
};  
  
// std::is_standard_layout<Derived> == true  
struct Derived : public Base  
{  
    int x;  
    int y;  
};
```

La clase derivada también sería un diseño estándar si `Base` tuviera los miembros de datos y `Derived` solo tuviera funciones miembro.

Tipos POD

Cuando una clase o estructura es tanto trivial como de diseño estándar, se trata de un tipo POD (Plain Old Data). El diseño de memoria de los tipos POD, por tanto, es contiguo y cada miembro tiene una dirección más alta que el miembro que se declaró antes, por lo que en estos tipos se pueden realizar copias byte a byte y E/S binaria. Los tipos escalares, como `int`, también son tipos POD. Los tipos POD que son clases pueden tener solo los tipos POD como miembros de datos no estáticos.

Ejemplo

En el ejemplo siguiente se muestran las diferencias entre los tipos trivial, de diseño estándar y POD:

```
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;
}
```

Tipos literales

Un tipo literal es aquel cuyo diseño se puede determinar en tiempo de compilación.

Estos son los tipos literales:

- void
- tipos escalares
- references
- Matrices de void, tipos escalares o referencias.
- Una clase que tiene un destructor trivial y uno o varios constructores constexpr que no son constructores de movimiento ni de copias. Además, todos sus miembros de datos no estáticos y sus clases base deben ser tipos literales y no volátiles.

Consulte también

[Conceptos básicos](#)

Clases C++ como tipos de valor

Artículo • 03/03/2023 • Tiempo de lectura: 4 minutos

Las clases de C++ son, de forma predeterminada, tipos de valor. Se pueden especificar como tipos de referencia, lo que permite el comportamiento polimórfico para admitir la programación orientada a objetos. A veces, los tipos de valor se ven desde la perspectiva del control de la memoria y el diseño, mientras que los tipos de referencia se refieren a clases base y funciones virtuales con fines polimórficos. De forma predeterminada, los tipos de valor se pueden copiar, lo que significa que siempre hay un constructor de copias y un operador de asignación de copia. Para los tipos de referencia, haga que la clase no se puede copiar (deshabilite el constructor de copia y el operador de asignación de copia) y use un destructor virtual, que admite su polimorfismo previsto. Los tipos de valor están también relacionados con los contenidos que, cuando se copian, le proporcionan siempre dos valores independientes que se pueden modificar por separado. Los tipos de referencia están relacionados con la identidad: ¿qué tipo de objeto es? Por este motivo, los "tipos de referencia" también se conocen como "tipos polimórficos".

Si realmente desea un tipo similar a una referencia (clase base, funciones virtuales), tiene que deshabilitar explícitamente el copiado, como se muestra en la clase `MyRefType` en el código siguiente.

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

La compilación del código anterior dará como resultado el siguiente error:

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'
```

Tipos de valor y eficiencia de movimiento

Se evita la sobrecarga de asignación de copias debido a las nuevas optimizaciones de copia. Por ejemplo, cuando inserta una cadena en medio de un vector de cadenas, no hay ninguna sobrecarga de reasignación de copia, solo un movimiento, incluso si da como resultado el crecimiento del propio vector. Estas optimizaciones también se aplican a otras operaciones: por ejemplo, realizando una operación de adición en dos objetos enormes. ¿Cómo se habilitan estas optimizaciones de operaciones de valor? El compilador los habilita implícitamente, al igual que el compilador puede generar automáticamente los constructores de copia. Sin embargo, la clase tiene que "participar" en las asignaciones de movimiento y los constructores de movimiento declarándolos en la definición de clase. El movimiento usa la referencia rvalue de doble & (&&) en las declaraciones de función miembro adecuadas y en la definición de los métodos del constructor de movimiento y de la asignación de movimiento. También debe insertar el código correcto para "vaciar" el objeto de origen.

¿Cómo decide si necesita habilitar las operaciones de movimiento? Si ya sabe que necesita habilitar la construcción de copia, probablemente quiera habilitar también la construcción de movimiento, especialmente si es más barato que una copia profunda. Sin embargo, si sabe que necesita compatibilidad con el movimiento, no significa necesariamente que quiera habilitar las operaciones de copia. Este último caso se denominaría "tipo de solo movimiento". Un ejemplo que ya está en la biblioteca estándar es `unique_ptr`. Como nota al margen, el antiguo `auto_ptr` está en desuso y fue reemplazado por `unique_ptr` precisamente debido a la falta de compatibilidad con la semántica de movimiento en la versión anterior de C++.

Mediante la semántica de transferencia de recursos, puede devolver por valor o insertar en medio. El movimiento es una optimización de la copia. No es necesaria la asignación de montón como solución alternativa. Considere el siguiente seudocódigo:

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

```

Habilitación del movimiento para los tipos de valor adecuados

Para una clase similar a un valor en la que el movimiento puede ser más barato que una copia profunda, habilite la construcción de movimiento y la asignación de movimiento para mejorar la eficacia. Considere el siguiente seudocódigo:

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!"); }
};
```

Si habilita la construcción o asignación de copia, habilite también la construcción o asignación de movimiento si puede ser más barato que una copia en profundidad.

Algunos tipos *que no son de valor* son de solo movimiento, como cuando no se puede clonar un recurso, solo se transfiere la propiedad. Ejemplo: `unique_ptr`.

Consulte también

[Sistema de tipos de C++](#)

[Aquí está otra vez C++](#)

[Referencia del lenguaje C++](#)

[Biblioteca estándar de C++](#)

Conversiones de tipos y seguridad de tipos

Artículo • 03/03/2023 • Tiempo de lectura: 9 minutos

En este documento se identifican problemas comunes de la conversión de tipos y se describe cómo evitarlos en el código de C++.

Cuando se escribe un programa de C++, es importante asegurarse de tener seguridad de tipos. Esto significa que cada variable, argumento de función y valor devuelto de una función almacena una clase aceptable de datos y que las operaciones que implican valores de distintos tipos “tienen sentido” y no provocan pérdida de datos, interpretaciones incorrectas de los patrones de bits o daños en la memoria. Un programa que nunca convierte valores de un tipo a otro de forma implícita o explícita tiene seguridad de tipos por definición. No obstante, las conversiones de tipos se requieren a veces, incluso las no seguras. Por ejemplo, podría ser necesario almacenar el resultado de una operación de punto flotante en una variable de tipo `int` o quizás se deba pasar el valor de un tipo `unsigned int` a una función que tome un tipo `signed int`. Ambos ejemplos ilustran las conversiones no seguras porque pueden producir pérdida de datos o la reinterpretación de un valor.

Cuando el compilador detecta una conversión no segura, emite un error o una advertencia. Un error detiene la compilación; una advertencia permite que la compilación continúe, pero indica un posible error en el código. Sin embargo, aunque el programa se compile sin advertencias, todavía pueden contener código que lleve a conversiones implícitas que generan resultados incorrectos. Pueden producirse errores de tipos en el código debido a las conversiones explícitas.

Conversiones de tipos implícitas

Cuando una expresión contiene operandos de diferentes tipos integrados y no hay conversiones explícitas presentes, el compilador usa las *conversiones estándar* integradas para convertir uno de los operandos de forma que los tipos coincidan. El compilador intenta las conversiones en una secuencia bien definida hasta que una sea correcta. Si la conversión seleccionada es una promoción, el compilador no emite una advertencia. Si la conversión es una restricción, el compilador emite una advertencia sobre la posible pérdida de datos. El hecho de que se produzca en efecto la pérdida de datos depende de los valores reales implicados, pero se recomienda tratar esta advertencia como un error. Si está implicado un tipo definido por el usuario, el compilador intenta utilizar las conversiones especificadas en la definición de clase. Si no

encuentra una conversión aceptable, el compilador emite un error y no se compila el programa. Para más información sobre las reglas que rigen las conversiones estándar, consulte [Conversiones estándar](#). Para más información sobre las conversiones definidas por el usuario, consulte [Conversiones definidas por el usuario \(C++/CLI\)](#).

Conversiones de ampliación (promoción)

En una conversión de ampliación, un valor de una variable menor se asigna a una variable mayor sin pérdida de datos. Dado que las conversiones de ampliación siempre son seguras, el compilador las realiza de forma silenciosa y no emite advertencias. Las conversiones siguientes son de ampliación.

| From | En |
|--|---|
| Cualquier tipo entero <code>signed</code> o <code>unsigned</code> , excepto <code>long long</code> o <code>__int64</code> | <code>double</code> |
| <code>bool</code> o <code>char</code> | Cualquier otro tipo integrado |
| <code>short</code> o <code>wchar_t</code> | <code>int</code> , <code>long</code> , <code>long long</code> |
| <code>int</code> , <code>long</code> | <code>long long</code> |
| <code>float</code> | <code>double</code> |

Conversiones de restricción (coerción)

El compilador realiza las conversiones de restricción implícitamente, pero le advierte sobre la pérdida de datos. Tome estas advertencias muy en serio. Si está seguro de que no se producirá ninguna pérdida de datos porque los valores de la variable mayor cabrán siempre en la variable menor, agregue una conversión explícita de modo que el compilador no emita ninguna advertencia más. Si no está seguro de que la conversión sea segura, agregue al código alguna clase de comprobación en tiempo de ejecución para controlar la posible pérdida de datos a fin de que no cause resultados incorrectos en el programa.

Cualquier conversión de un tipo de punto flotante a un tipo entero es una conversión de restricción porque la parte fraccionaria del valor de punto flotante se descarta y se pierde.

El ejemplo de código siguiente muestra algunas conversiones de restricción implícitas y las advertencias correspondientes que emite el compilador.

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
```

Conversiones con y sin signo

Un tipo entero con signo y su homólogo sin signo son siempre del mismo tamaño, aunque difieren en cuanto a la forma de interpretar el patrón de bits para la transformación del valor. El ejemplo de código siguiente muestra lo que ocurre cuando el mismo patrón de bits se interpreta como valor con signo y como valor sin signo. El patrón de bits almacenado en `num` y `num2` nunca cambia respecto a lo que se muestra en la ilustración anterior.

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"
```

Observe que los valores se reinterpretan en ambas direcciones. Si el programa produce resultados extraños en los que el signo del valor parece lo contrario de lo esperado, busque las conversiones implícitas entre los tipos enteros con y sin signo. En el ejemplo siguiente, el resultado de la expresión `(0 - 1)` se convierte implícitamente de `int` a `unsigned int` cuando se almacena en `num`. Esto hace que el patrón de bits se reinterprete.

C++

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

El compilador no advierte sobre las conversiones implícitas entre los tipos enteros con y sin signo. Por tanto, se recomienda evitar por completo las conversiones de tipos con o sin signo. Si no puede evitarlas, agregue una comprobación en tiempo de ejecución para detectar si el valor que se convierte es mayor o igual que cero y menor o igual que el valor máximo del tipo con signo. Los valores de este intervalo se transferirán de tipos con signo a tipos sin signo, o viceversa, sin reinterpretaciones.

Conversiones de puntero

En muchas expresiones, una matriz de estilo C se convierte implícitamente a un puntero al primer elemento de la matriz y las conversiones de constantes pueden ocurrir de forma silenciosa. Aunque esto es conveniente, también es potencialmente propenso a errores. Por ejemplo, el ejemplo de código mal diseñado siguiente es absurdo, pero se compilará y genera un resultado de "p". Primero, el literal de la constante de cadena "Ayuda" se convierte en `char*`, que señala al primer elemento de la matriz; ese puntero se incrementa en tres elementos para que señale al último elemento "p".

C++

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

Conversiones explícitas

Mediante una operación de conversión, puede indicar al compilador que convierta un valor de un tipo a otro tipo. El compilador generará un error en algunos casos si los dos tipos no tienen ninguna relación entre sí, pero en otros casos no generará un error aunque la operación no tenga seguridad de tipos. Utilice las conversiones con moderación porque cualquier conversión de un tipo a otro es un origen potencial de errores del programa. Sin embargo, las conversiones son necesarias a veces y no todas son igualmente peligrosas. Una conversión se usa eficazmente cuando el código realiza una conversión de restricción y se sabe que la conversión no causará resultados incorrectos en el programa. En efecto, esto indica al compilador que sabe lo que está haciendo y que deje de molestarle con advertencias sobre ello. Otro uso es convertir desde una clase de puntero a derivado a una clase de puntero a base. Otro uso es desechar la declaración como `const` de una variable para pasarla a una función que

requiera un argumento que no es const. La mayoría de estas operaciones de conversión implican algunos riesgos.

En la programación de estilo C, se utiliza el mismo operador de conversión de estilo C para todos los tipos de conversiones.

C++

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

El operador de conversión de estilo C es idéntico al operador de llamada () y, por consiguiente, no sobresale en el código y es sencillo pasarlo por alto. Ambos presentan problemas porque son difíciles de reconocer de un vistazo o de buscar, y son lo bastante dispares como para invocar cualquier combinación de `static`, `const` y `reinterpret_cast`. Averiguar lo que hace realmente una conversión de estilo antiguo puede ser difícil y propenso a errores. Por todas estas razones, cuando se requiere una conversión, recomendamos utilizar uno de los siguientes operadores de conversión de C++, que en algunos casos tienen mucha más seguridad de tipos y expresan mucho más explícitamente la intención de la programación:

- `static_cast`, para las conversiones que se comprueban solo en tiempo de compilación. `static_cast` devuelve un error si el compilador detecta que intenta realizar conversiones entre tipos que son totalmente incompatibles. También puede utilizarlo para convertir entre puntero a base y puntero a derivado, pero el compilador no puede determinar siempre si tales conversiones son seguras en tiempo de ejecución.

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);
```

Para obtener más información, vea [static_cast](#).

- `dynamic_cast`, para conversiones seguras comprobadas en tiempo de ejecución entre puntero a base y puntero a derivado. Una conversión `dynamic_cast` es más

segura que una conversión `static_cast` para las conversiones de restricción, pero la comprobación en tiempo de ejecución implica cierta sobrecarga.

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;
```

Para obtener más información, vea [dynamic_cast](#).

- `const_cast`, para desechar la declaración de `const` de una variable o convertir una variable no `const` en `const`. Desechar la declaración de `const` mediante este operador es tan propenso a errores como usar una conversión de estilo C, excepto que con `const_cast` es menos probable realizar la conversión accidentalmente. A veces, es necesario desechar la declaración de `const` de una variable, por ejemplo, para pasar una variable `const` a una función que toma un parámetro que no es `const`. El ejemplo siguiente muestra cómo hacerlo.

C++

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

Para obtener más información, vea [const_cast](#).

- `reinterpret_cast`, para conversiones entre tipos no relacionados, como un tipo de puntero y un tipo `int`.

ⓘ Nota

Este operador de conversión no se usa tan a menudo como los demás y no se garantiza que sea portable a otros compiladores.

En el ejemplo siguiente se muestra la diferencia entre `reinterpret_cast` y `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.
```

Para más información, consulte [Operador reinterpret_cast](#).

Consulte también

[Sistema de tipos de C++](#)

[Aquí está otra vez C++](#)

[Referencia del lenguaje C++](#)

[Biblioteca estándar de C++](#)

Conversiones estándar

Artículo • 03/03/2023 • Tiempo de lectura: 14 minutos

El lenguaje C++ define conversiones entre sus tipos fundamentales. También define conversiones para tipos derivados de puntero, referencia y puntero a miembro. Estas conversiones se denominan *conversiones estándar*.

Esta sección trata las conversiones estándar siguientes:

- Promociones de enteros
- Conversiones de enteros
- Conversiones de punto flotante
- Conversiones de punto flotante y de enteros
- Conversiones aritméticas
- Conversiones de puntero
- Conversiones de referencia
- Conversiones de puntero a miembro

⚠ Nota

Los tipos definidos por el usuario pueden especificar sus propias conversiones. La conversión de tipos definidos por el usuario se trata en [Constructores y Conversiones](#).

El código siguiente provoca conversiones (en este ejemplo, promociones de enteros):

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;
```

El resultado de una conversión solo es un valor L si genera un tipo de referencia. Por ejemplo, una conversión definida por el usuario declarada como `operator int&()` devuelve una referencia y es un valor L, mientras que una conversión declarada como `operator int()` devuelve un objeto y no es un valor L.

Promociones de enteros

Los objetos de un tipo entero se pueden convertir a otro tipo entero más amplio, es decir, un tipo que puede representar un conjunto de valores más grande. Este tipo de ampliación de conversión se denomina *promoción de entero*. Con la promoción de entero, puede usar los tipos siguientes en una expresión dondequiera que se pueda usar otro tipo entero:

- Objetos, literales y constantes de tipo `char` y `short int`
- Tipos de enumeración
- Campos de bits `int`
- Enumerators

Las promociones de C++ tienen la cualidad de "conservación de valores", es decir, se garantiza que el valor después de la promoción es igual que el valor antes de la promoción. En las promociones que conservan los valores, los objetos de tipos enteros más cortos (tales como campos de bits u objetos de tipo `char`) se promueven al tipo `int` si `int` puede representar el intervalo completo del tipo original. Si `int` no puede representar el intervalo completo de valores, el objeto se promueve al tipo `unsigned int`. Aunque esta estrategia es la misma que la usada por el estándar C, las conversiones que poseen la cualidad de conservación de valores no conservan el "tipo signed/unsigned" del objeto.

Las promociones que poseen la cualidad de conservación de valores y las promociones que conservan el tipo signed/unsigned generan normalmente los mismos resultados. Aun así, pueden generar resultados diferentes si el objeto que se promueve aparece como:

- Un operando de `/`, `%`, `/=`, `%=`, `<`, `<=`, `>` o `>=`

Estos operadores dependen del signo para determinar el resultado. Las promociones que poseen la cualidad de conservación de valores y que conservan el tipo signed/unsigned generan resultados diferentes cuando se aplican a estos operandos.

- El operando izquierdo de `>>` o `>>=`

Estos operadores tratan las cantidades signed y unsigned de forma diferente en una operación de desplazamiento. En el caso de cantidades signed, la operación de desplazamiento a la derecha hace que el bit de signo se propague a las posiciones de bits desocupadas, mientras que las posiciones de bits desocupadas se rellenan con ceros en el caso de las cantidades unsigned.

- Un argumento a una función sobrecargada o el operando de un operador sobrecargado que depende de la condición signed/unsigned del tipo de operando para la coincidencia de argumentos. Para obtener más información sobre la definición de operadores sobrecargados, consulte [Sobrecarga de operadores](#).

Conversiones de enteros

Las *conversiones de enteros* son conversiones entre tipos enteros. Los tipos enteros son `char`, `short` (o `short int`), `int`, `long` y `long long`. Estos tipos se pueden calificar con `signed` o `unsigned`, y `unsigned` solo se puede usar como una abreviatura para `unsigned int`.

De con signo a sin signo

Los objetos de tipos enteros con signo se pueden convertir en los tipos sin signo correspondientes. Cuando se produce esta conversión, el patrón de bits real no cambia, pero sí la interpretación de los datos. Considere este código:

C++

```
#include <iostream>

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

    cout << (u = i) << "\n";
}
```

// Output: 65533

En el ejemplo anterior, se define `i signed short` y se inicializa en un número negativo. La expresión `(u = i)` hace que `i` se convierta en `unsigned short` antes de la asignación a `u`.

De con signo a sin signo

Los objetos de tipos enteros sin signo se pueden convertir en los tipos con signo correspondientes. Pero si el valor sin signo está fuera del intervalo representable del tipo con signo, el resultado no tendrá el valor correcto, como se muestra en el ejemplo siguiente:

C++

```
#include <iostream>

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

    cout << (i = u) << "\n";
}
```

//Output: -3

En el ejemplo anterior, `u` es un objeto entero `unsigned short` que se debe convertir en una cantidad con signo para evaluar la expresión `(i = u)`. Como su valor no se puede representar correctamente en `signed short`, los datos se interpretan de forma incorrecta como se muestra.

Conversiones de punto flotante

Un objeto de tipo flotante se puede convertir de forma segura en un tipo flotante más preciso; es decir, la conversión no provoca ninguna pérdida de significado. Por ejemplo, las conversiones de `float` a `double` o de `double` a `long double` son seguras, y el valor no cambia.

Un objeto de tipo flotante se puede convertir en un tipo menos preciso, si se encuentra en un intervalo representable por ese tipo. (Consulte [Límites flotantes](#) para conocer los intervalos de tipos de punto flotante). Si el valor original no se puede representar con exactitud, se puede convertir en el siguiente valor representable mayor o menor. Si no existe dicho valor, el resultado es indefinido. Considere el ejemplo siguiente:

C++

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

El valor máximo que se puede representar mediante el tipo `float` es 3,402823466E38, un número mucho menor que 1E300. Por tanto, el número se convierte a infinito y el resultado es "inf".

Conversiones entre tipos integrales y de punto flotante

Algunas expresiones pueden producir la conversión de objetos de tipo flotante a tipos enteros, o viceversa. Cuando un objeto de tipo entero se convierte a un tipo flotante y el valor original no se puede representar exactamente, el resultado es el siguiente valor más alto o más bajo que se puede representar.

Cuando un objeto de tipo flotante se convierte en un tipo entero, la parte fraccionaria se *trunca* o se redondea hacia cero. Un número como 1,3 se convierte en 1, y -1,3 se convierte en -1. Si el valor truncado es mayor que el valor representable más alto o menor que el valor representable más bajo, el resultado es indefinido.

Conversiones aritméticas

Muchos operadores binarios (descritos en [Expresiones con operadores binarios](#)) originan la conversión de operandos y producen resultados de la misma forma. Las conversiones que estos operadores producen se denominan *conversiones aritméticas usuales*. Las conversiones aritméticas de operandos que tienen tipos nativos diferentes se realizan como se muestra en la tabla siguiente. Los tipos `typedef` se comportan de acuerdo con sus tipos nativos subyacentes.

Condiciones para la conversión de tipos

| Condiciones satisfechas | Conversión |
|---|--|
| Uno de los operandos es de tipo <code>long double</code> . | El otro operando se convierte al tipo <code>long double</code> . |
| La condición anterior no se cumple y uno de los operandos es de tipo <code>double</code> . | El otro operando se convierte al tipo <code>double</code> . |
| Las condiciones anteriores no se cumplen y uno de los operandos es de tipo <code>float</code> . | El otro operando se convierte al tipo <code>float</code> . |

| Condiciones satisfechas | Conversión |
|---|--|
| Las condiciones anteriores no se satisfacen (ninguno de los operandos es de tipo flotante). | <p>Los operandos obtienen promociones de enteros de la siguiente manera:</p> <ul style="list-style-type: none"> - Si alguno de los operandos es de tipo <code>unsigned long</code>, el otro operando se convierte al tipo <code>unsigned long</code>. - Si no se cumple la condición anterior y alguno de los operandos es de tipo <code>long</code> y el otro de tipo <code>unsigned int</code>, ambos operandos se convierten al tipo <code>unsigned long</code>. - Si las dos condiciones anteriores no se cumplen y alguno de los operandos es de tipo <code>long</code>, el otro operando se convierte al tipo <code>long</code>. - Si las tres condiciones anteriores no se cumplen y alguno de los operandos es de tipo <code>unsigned int</code>, el otro operando se convierte al tipo <code>unsigned int</code>. - Si no se cumple ninguna de las condiciones anteriores, ambos operandos se convierten al tipo <code>int</code>. |

En el código siguiente se muestran las reglas de conversión descritas en la tabla:

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;
}
```

La primera instrucción del ejemplo anterior muestra la multiplicación de dos tipos enteros, `iVal` y `ulVal`. La condición que se cumple es que ninguno de los operandos es de tipo flotante y un operando es de tipo `unsigned int`. Así pues, el otro operando (`iVal`) se convierte al tipo `unsigned int`. Después, el resultado se asigna a `dVal`. La condición que se cumple en este caso es que un operando es de tipo `double`, de modo que el resultado `unsigned int` de la multiplicación se convierte al tipo `double`.

La segunda instrucción del ejemplo anterior muestra la suma de un `float` y un tipo entero: `fVal` y `ulVal`. La variable `ulVal` se convierte al tipo `float` (la tercera condición

de la tabla). El resultado de la suma se convierte al tipo `double` (la segunda condición de la tabla) y se asigna a `dVal`.

Conversiones de puntero

Los punteros se pueden convertir durante la asignación, inicialización, comparación y en otras expresiones.

De puntero a clases

Hay dos casos en los que un puntero a una clase se puede convertir en un puntero a una clase base.

El primer caso es cuando la clase base especificada es accesible y la conversión es inequívoca. Para obtener más información sobre las referencias de clase base ambigua, consulte [Varias clases base](#).

Si una clase base es accesible depende del tipo de herencia utilizada en la derivación. Considere la herencia que se muestra en la siguiente ilustración.

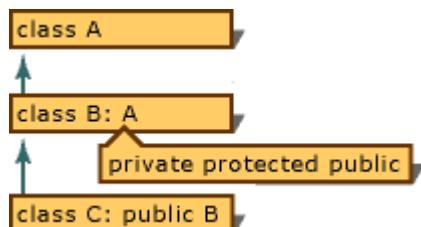


Gráfico de herencia para ilustrar la accesibilidad de clase base

En la tabla siguiente se muestra la accesibilidad de la clase base para la situación que se muestra en la ilustración.

| Tipo de función | Derivación | Conversión de |
|---|------------|---------------|
| | | B* a A* legal |
| Función externa (no de ámbito de clase) | Privada | No |
| | Protegido | No |
| | Público | Sí |
| Función miembro B (en ámbito B) | Privados | Sí |
| | Protegido | Sí |
| | Público | Sí |

| Tipo de función | Derivación | Conversión de |
|---------------------------------|------------|---------------|
| | | B* a A* legal |
| Función miembro C (en ámbito C) | Privada | No |
| | Protegido | Sí |
| | Público | Sí |

El segundo caso en el que un puntero a una clase se puede convertir a un puntero a una clase base es cuando se utiliza una conversión de tipos explícita. Para obtener más información sobre las conversiones de tipo explícitas, consulte [Operador de conversión explícita de tipos](#).

El resultado de tal conversión es un puntero al *subobjeto*, es decir, la parte del objeto que la clase base describe completamente.

En el código siguiente se definen dos clases, `A` y `B`, donde `B` se deriva de `A`. (Para obtener más información sobre la herencia, consulte [Clases derivadas](#)). Después, define `bObject`, un objeto de tipo `B` y dos punteros (`pA` y `pB`) que apuntan al objeto.

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
}
```

El puntero `pA` es de tipo `A *`, lo que se puede interpretar como "puntero a un objeto de tipo `A`". Los miembros de `bObject` (como `BComponent` y `BMemberFunc`) son únicos para el tipo `B` y, por lo tanto, inaccesibles mediante `pA`. El puntero `pA` solo permite el acceso a esas características (funciones de miembro y datos) del objeto que se definen en la clase `A`.

De puntero a función

Un puntero a una función puede convertirse al tipo `void *`, si el tipo `void *` es lo bastante grande para contener ese puntero.

De puntero a void

Los punteros al tipo `void` se pueden convertir a punteros a cualquier otro tipo, pero solo con una conversión de tipo explícita (a diferencia de C). Un puntero a cualquier tipo se puede convertir implícitamente en un puntero al tipo `void`. Un puntero a un objeto incompleto de un tipo se puede convertir a un puntero a `void` (implícito) y viceversa (explícitamente). El resultado de dicha conversión es igual al valor del puntero original. Un objeto se considera incompleto si se declara, pero no hay suficiente información disponible para determinar su tamaño o clase base.

Un puntero a cualquier objeto que no sea `const` o `volatile` se puede convertir implícitamente en un puntero de tipo `void *`.

punteros const y volatile

C++ no proporciona una conversión estándar de un tipo `const` o `volatile` a un tipo que no es `const` o `volatile`. Sin embargo, se puede especificar cualquier tipo de conversión mediante conversiones de tipos explícitas (incluidas las conversiones que no son seguras).

ⓘ Nota

Los punteros a miembros de C++, excepto los punteros a miembros estáticos, son diferentes de los punteros normales y no tienen las mismas conversiones estándar. Los punteros a miembros estáticos son punteros normales y tienen las mismas conversiones que los punteros normales.

conversiones de puntero nulo

Una expresión constante entera que se evalúa como cero, o una conversión de expresión a un tipo de puntero, se convierte a un puntero denominado *puntero nulo*. Este puntero siempre será diferente de un puntero a cualquier objeto o función válido. Una excepción son los punteros a objetos basados, que pueden tener el mismo desplazamiento y seguir apuntando a objetos diferentes.

En C++11, debe preferirse el tipo `nullptr` al puntero nulo de estilo C.

Conversiones de expresiones de puntero

Cualquier expresión con un tipo de matriz se puede convertir a un puntero del mismo tipo. El resultado de la conversión es un puntero al primer elemento de matriz. El ejemplo siguiente muestra este tipo de conversión:

C++

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

Una expresión que da lugar a una función que devuelve un tipo determinado se convierte a un puntero a una función que devuelve ese tipo, excepto cuando:

- La expresión se usa como operando para el operador address-of (&).
- La expresión se utiliza como operando para el operador de llamada a función.

Conversiones de referencia

Una referencia a una clase se puede convertir en una referencia a una clase base en estos casos:

- Se puede acceder a la clase base especificada.
- La conversión no es ambigua. (Para obtener más información sobre las referencias de clase base ambigua, consulte [Varias clases base](#)).

El resultado de la conversión es un puntero al subobjeto que representa la clase base.

Puntero a miembro

Los punteros a miembros de clase se pueden convertir durante la asignación, la inicialización, la comparación y otras expresiones. En esta sección se describen las siguientes conversiones de puntero a miembro:

Puntero a miembro de clase base

Un puntero a un miembro de una clase base se puede convertir en un puntero a un miembro de una clase derivada de esa clase base cuando se cumplen las condiciones siguientes:

- Se tiene acceso a la conversión inversa, desde el puntero a la clase derivada al puntero de la clase base.
- La clase derivada no hereda virtualmente de la clase base.

Cuando el operando izquierdo es un puntero a miembro, el operando derecho debe ser un tipo de puntero a miembro o una expresión constante que se evalúe como 0. Esta asignación solo es válida en los casos siguientes:

- El operando derecho es un puntero a un miembro de la misma clase que el operando izquierdo.
- El operando izquierdo es un puntero a un miembro de una clase que se ha derivado de forma pública e inequívoca de la clase del operando derecho.

Conversiones de puntero nulo a miembro

Una expresión constante entera que se evalúa como cero se convierte a un puntero nulo. Este puntero siempre será diferente de un puntero a cualquier objeto o función válido. Una excepción son los punteros a objetos basados, que pueden tener el mismo desplazamiento y seguir apuntando a objetos diferentes.

El código siguiente muestra la definición de un puntero al miembro `i` en la clase `A`. El puntero, `pai`, se inicializa en 0, que es el puntero null.

C++

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

int A::*pai = 0;

int main()
{}
```

Consulte también

[Referencia del lenguaje C++](#)

Tipos integrados (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 6 minutos

Los tipos integrados (también denominados *tipos fundamentales*) se especifican mediante el estándar del lenguaje C++ y están integrados en el compilador. Los tipos integrados no se definen en ningún archivo de encabezado. Los tipos integrados se dividen en tres categorías principales: *entero*, *punto flotante* y *void*. Los tipos de datos enteros representan números enteros. Los tipos de punto flotante pueden especificar valores que podrían tener partes fraccionarias. El compilador trata la mayoría de los tipos integrados como tipos distintos. Aun así, algunos tipos son *sinónimos*, o el compilador los trata como tipos equivalentes.

Un tipo void

El tipo `void` describe un conjunto de valores vacío. No se puede especificar ninguna variable de tipo `void`. El tipo `void` se usa principalmente para declarar funciones que no devuelven ningún valor o para declarar punteros genéricos a datos sin tipo o con un tipo arbitrario. Cualquier expresión se puede convertir explícitamente al tipo `void`. Sin embargo, estas expresiones se limitan a los siguientes usos:

- Una instrucción de expresión. (Para obtener más información, consulte [Expresiones](#)).
- El operando izquierdo del operador de coma. (Para obtener más información, consulte [Operador de coma](#)).
- El segundo o tercer operando del operador condicional (`? :`). (Para obtener más información, consulte [Expresiones con el operador condicional](#)).

std::nullptr_t

La palabra clave `nullptr` es una constante de puntero null de tipo `std::nullptr_t`, que se puede convertir a cualquier tipo de puntero sin formato. Para más información, consulte [nullptr](#).

Tipo booleano

El tipo `bool` puede tener valores `true` y `false`. El tamaño del tipo `bool` es específico de la implementación. Consulte [Tamaños de tipos integrados](#) para obtener detalles de

implementación específicos de Microsoft.

Tipos de caracteres

El tipo `char` es un tipo de representación de caracteres que codifica eficazmente los miembros del juego básico de caracteres de ejecución. El compilador de C++ trata las variables del tipo `char`, `signed char` y `unsigned char` como si tuvieran tipos diferentes.

Específico de Microsoft: las variables de tipo `char` se promueven a `int` como si fueran del tipo `signed char` de forma predeterminada, a menos que se use la opción de compilación `/J`. En ese caso se tratan como de tipo `unsigned char` y se promueven a `int` sin la extensión de signo.

Una variable de tipo `wchar_t` es un carácter ancho o multibyte. Use el prefijo `L` delante de un carácter o un literal de cadena para especificar el tipo de carácter ancho.

Específico de Microsoft: de forma predeterminada, `wchar_t` es un tipo nativo, pero se puede usar `/Zc:wchar_t-` para convertir `wchar_t` en una definición de tipo para `unsigned short`. El `_wchar_t` tipo es un sinónimo específico de Microsoft para el tipo `wchar_t` nativo.

El tipo `char8_t` se usa para la representación de caracteres UTF-8. Tiene la misma representación que `unsigned char`, pero el compilador lo trata como un tipo distinto. El tipo `char8_t` es nuevo en C++20. **Específico de Microsoft:** el uso de `char8_t` requiere la opción del compilador `/std:c++20` o una versión posterior (como `/std:c++latest`).

El tipo `char16_t` se usa para la representación de caracteres UTF-16. Debe ser lo suficientemente grande para representar cualquier unidad de código UTF-16. El compilador lo trata como un tipo distinto.

El tipo `char32_t` se usa para la representación de caracteres UTF-32. Debe ser lo suficientemente grande para representar cualquier unidad de código UTF-32. El compilador lo trata como un tipo distinto.

Tipos de punto flotante

Los tipos de punto flotante usan una representación IEEE-754 para proporcionar una aproximación de valores fraccionarios en una amplia gama de magnitudes. En la tabla siguiente se muestran los tipos de punto flotante de C++ y las restricciones comparativas en tamaños de tipo de punto flotante. Estas restricciones son obligatorias por el estándar de C++ y son independientes de la implementación de Microsoft. El

tamaño absoluto de los tipos de punto flotante integrados no se especifica en el estándar.

| Tipo | Contenido |
|--------------------------|---|
| <code>float</code> | El tipo <code>float</code> es el tipo de punto flotante más pequeño de C++. |
| <code>double</code> | El tipo <code>double</code> es un tipo flotante superior o igual al tipo <code>float</code> , pero inferior o igual al tamaño del tipo <code>long double</code> . |
| <code>long double</code> | El tipo <code>long double</code> es un tipo de punto flotante que es superior o igual al tipo <code>double</code> . |

Específico de Microsoft: la representación de `long double` y `double` es idéntica. Aun así, el compilador trata `long double` y `double` como tipos distintos. El compilador de Microsoft C++ usa las representaciones de punto flotante de 4 y 8 bytes conforme a IEEE-754. Para obtener más información, consulte [Representación de punto flotante IEEE](#).

Tipos enteros

El tipo `int` es el tipo entero básico predeterminado. Puede representar todos los números enteros en un intervalo específico de la implementación.

Una representación de entero *con signo* es aquella que puede contener valores positivos y negativos. Se usa de forma predeterminada o cuando está presente la palabra clave modificadora `signed`. La palabra clave modificadora `unsigned` especifica una representación *sin signo* que solo puede contener valores no negativos.

Un modificador de tamaño especifica el ancho en bits de la representación de entero que se usa. El lenguaje admite los modificadores `short`, `long` y `long long`. Un tipo `short` debe tener al menos 16 bits de ancho. Un tipo `long` debe tener al menos 32 bits de ancho. Un tipo `long long` debe tener al menos 64 bits de ancho. El estándar especifica una relación de tamaño entre los tipos enteros:

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

Una implementación debe mantener los requisitos de tamaño mínimo y la relación de tamaño para cada tipo. Aun así, los tamaños reales pueden variar entre implementaciones. Consulte [Tamaños de tipos integrados](#) para obtener detalles de implementación específicos de Microsoft.

La palabra clave `int` puede omitirse cuando se especifican `signed`, `unsigned` o modificadores de tamaño. Los modificadores y el tipo `int` pueden aparecer en cualquier orden, si están presentes. Por ejemplo, `short unsigned` y `unsigned int short` hacen referencia al mismo tipo.

Sinónimos de tipos enteros

El compilador considera sinónimos los siguientes grupos de tipos:

- `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`

Los tipos enteros específicos de Microsoft incluyen los tipos `_int8`, `_int16`, `_int32` y `_int64` de ancho específico. Estos tipos pueden usar los modificadores `signed` y `unsigned`. El tipo de datos `_int8` es sinónimo del tipo `char`, `_int16` es sinónimo del tipo `short`, `_int32` es sinónimo del tipo `int` y `_int64` es sinónimo del tipo `long long`.

Tamaños de los tipos integrados

La mayoría de los tipos integrados tienen tamaños definidos por la implementación. En la tabla siguiente se muestra la cantidad de almacenamiento necesaria para los tipos integrados de Microsoft C++. En concreto, `long` es de 4 bytes incluso en sistemas operativos de 64 bits.

| Tipo | Size |
|--|---------|
| <code>bool</code> , <code>char</code> , <code>char8_t</code> , <code>unsigned char</code> , <code>signed char</code> , <code>_int8</code> | 1 byte |
| <code>char16_t</code> , <code>_int16</code> , <code>short</code> , <code>unsigned short</code> , <code>wchar_t</code> , <code>_wchar_t</code> | 2 bytes |
| <code>char32_t</code> , <code>float</code> , <code>_int32</code> , <code>int</code> , <code>unsigned int</code> , <code>long</code> , <code>unsigned long</code> | 4 bytes |

| Tipo | Size |
|--|---------|
| <code>double</code> , <code>__int64</code> , <code>long double</code> , <code>long long</code> , <code>unsigned long long</code> | 8 bytes |

Consulte [Intervalos de tipo de datos](#) para obtener un resumen del intervalo de valores de cada tipo.

Para obtener más información sobre la conversión de tipos, consulte [Conversiones estándar](#).

Consulte también

[Intervalos de tipo de datos](#)

Intervalos de tipo de datos

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Los compiladores de 32 y 64 bits de Microsoft C++ reconocen los tipos de la tabla que se incluye más adelante en este artículo.

- `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)`

Si su nombre comienza con dos caracteres de subrayado (`_`), un tipo de datos no es estándar.

Los intervalos que se especifican en la tabla siguiente son inclusivo-inclusivo.

| Nombre del tipo | Bytes | Otros nombres | Intervalo de valores |
|------------------------------|-------|---|-----------------------------------|
| <code>int</code> | 4 | <code>signed</code> | De -2.147.483.648 a 2.147.483.647 |
| <code>unsigned int</code> | 4 | <code>unsigned</code> | De 0 a 4.294.967.295 |
| <code>_int8</code> | 1 | <code>char</code> | De -128 a 127 |
| <code>unsigned _int8</code> | 1 | <code>unsigned char</code> | De 0 a 255 |
| <code>_int16</code> | 2 | <code>short, short int, signed short int</code> | De -32 768 a 32 767 |
| <code>unsigned _int16</code> | 2 | <code>unsigned short, unsigned short int</code> | De 0 a 65.535 |
| <code>_int32</code> | 4 | <code>signed, signed int, int</code> | De -2.147.483.648 a 2.147.483.647 |

| Nombre del tipo | Bytes | Otros nombres | Intervalo de valores |
|---------------------------------|-------------------------------|---|---|
| <code>unsigned __int32</code> | 4 | <code>unsigned, unsigned int</code> | De 0 a 4.294.967.295 |
| <code>__int64</code> | 8 | <code>long long, signed long long</code> | De -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807 |
| <code>unsigned __int64</code> | 8 | <code>unsigned long long</code> | De 0 a 18.446.744.073.709.551.615 |
| <code>bool</code> | 1 | None | <code>false</code> o <code>true</code> |
| <code>char</code> | 1 | None | De -128 a 127 de manera predeterminada |
| | | | De 0 a 255 cuando se compila mediante <code>/J</code> |
| <code>signed char</code> | 1 | None | De -128 a 127 |
| <code>unsigned char</code> | 1 | None | De 0 a 255 |
| <code>short</code> | 2 | <code>short int, signed short int</code> | De -32 768 a 32 767 |
| <code>unsigned short</code> | 2 | <code>unsigned short int</code> | De 0 a 65.535 |
| <code>long</code> | 4 | <code>long int, signed long int</code> | De -2.147.483.648 a 2.147.483.647 |
| <code>unsigned long</code> | 4 | <code>unsigned long int</code> | De 0 a 4.294.967.295 |
| <code>long long</code> | 8 | Ninguno (pero equivalente a <code>__int64</code>) | De -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807 |
| <code>unsigned long long</code> | 8 | Ninguno (pero equivalente a <code>unsigned __int64</code>) | De 0 a 18.446.744.073.709.551.615 |
| <code>enum</code> | Varía | None | |
| <code>float</code> | 4 | None | 3,4E +/- 38 (7 dígitos) |
| <code>double</code> | 8 | None | 1,7E +/- 308 (15 dígitos) |
| <code>long double</code> | Igual que <code>double</code> | None | Igual que <code>double</code> . |

| Nombre del tipo | Bytes | Otros nombres | Intervalo de valores |
|----------------------|-------|-----------------------|----------------------|
| <code>wchar_t</code> | 2 | <code>_wchar_t</code> | De 0 a 65.535 |

Dependiendo de cómo se use, una variable `_wchar_t` designa un tipo de caracteres anchos o multibyte. Utilice el prefijo `L` delante de un carácter o constante de cadena para designar la constante de tipo de carácter ancho.

`signed` y `unsigned` son modificadores que se puede utilizar con cualquier tipo entero excepto `bool`. Observe que `char`, `signed char` y `unsigned char` son tres tipos distintos para mecanismos como sobrecargas y plantillas.

Los tipos `int` y `unsigned int` tienen un tamaño de cuatro bytes. Sin embargo, el código portable no debe depender del tamaño de `int`, porque el estándar del lenguaje permite que sea específico de la implementación.

C/C++ en Visual Studio también admite tipos enteros con tamaño. Para obtener más información, consulte `_int8`, `_int16`, `_int32`, `_int64` y [Límites de enteros](#).

Para obtener más información sobre las restricciones de tamaño de cada tipo, consulte [Tipos integrados](#).

El intervalo de tipos enumerados varía dependiendo del contexto del lenguaje y de las marcas del compilador especificadas. Para obtener más información, vea [Declaraciones de enumeración de C](#) y [Enumeraciones](#).

Consulte también

[Palabras clave](#)

[Tipos integrados](#)

nullptr

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

La palabra clave `nullptr` especifica una constante de puntero nulo de tipo `std::nullptr_t`, que se puede convertir a cualquier tipo de puntero sin formato. Aunque puede usar la palabra clave `nullptr` sin incluir ningún encabezado, si el código usa el tipo `std::nullptr_t`, debe definirlo incluyendo el encabezado `<cstddef>`.

ⓘ Nota

La palabra clave `nullptr` también se define en C++/CLI para aplicaciones de código administrado y no es intercambiable con la palabra clave de C++ del estándar ISO. Si el código podría compilarse con la opción del compilador `/clr`, destinada a código administrado, use `_nullptr` en cualquier línea de código donde debe garantizar que el compilador usa la interpretación de C++ nativa. Para obtener más información, consulte [nullptr \(C++/CLI y C++/CX\)](#).

Comentarios

Evite usar `NULL` o cero (`0`) como constante de puntero nulo; `nullptr` es menos vulnerable en caso de mal uso y funciona mejor en la mayoría de las situaciones. Por ejemplo, dado `func(std::pair<const char *, double>)`, la llamada a `func(std::make_pair(NULL, 3.14))` produce un error del compilador. La macro `NULL` se expande a `0`, de modo que la llamada `std::make_pair(0, 3.14)` devuelve `std::pair<int, double>`, que no se puede convertir al tipo de parámetro `std::pair<const char *, double>` de `func`. La llamada a `func(std::make_pair(nullptr, 3.14))` se compila correctamente porque `std::make_pair(nullptr, 3.14)` devuelve `std::pair<std::nullptr_t, double>`, que se puede convertir en `std::pair<const char *, double>`.

Consulte también

[Palabras clave](#)

[nullptr \(C++/CLI y C++/CX\)](#)

void (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Cuando se usa como un tipo de valor devuelto de función, la palabra clave `void` especifica que la función no devuelve ningún valor. Cuando se usa para la lista de parámetros de una función, `void` especifica que la función no toma ningún parámetro. Cuando se usa en la declaración de un puntero, `void` especifica que el puntero es "universal".

Si el tipo de puntero es `void*`, el puntero puede apuntar a cualquier variable que no se declare con la palabra clave `const` o `volatile`. Un puntero `void*` no se puede desreferenciar a menos que se convierta en otro tipo. Un puntero `void*` se puede convertir en cualquier otro tipo de puntero de datos.

En C++, un puntero `void` puede apuntar a una función libre (una función que no es miembro de una clase) o a una función miembro estática, pero no a una función miembro no estática.

No se puede declarar una variable de tipo `void`.

Como cuestión de estilo, las directrices básicas de C++ recomiendan no usar `void` para especificar una lista de parámetros formales vacía. Para obtener más información, vea [Directrices básicas de C++ NL.25: No usar void como un tipo de argumento ↗](#).

Ejemplo

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;  
}
```

Consulte también

[Palabras clave](#)

[Tipos integrados](#)

bool (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Esta palabra clave es un tipo integrado. Una variable de este tipo puede tener valores `true` y `false`. Las expresiones condicionales tienen el tipo `bool` y, por lo tanto, tienen valores de tipo `bool`. Por ejemplo, `i != 0` ahora tiene `true` o `false` en función del valor de `i`.

Visual Studio 2017, versión 15.3 y posteriores (disponible con `/std:c++17` y versiones posteriores): es posible que el operando de un operador de incremento o decremento de prefijo o de postfijo no sea de tipo `bool`. En otras palabras, dada una variable `b` de tipo `bool`, estas expresiones ya no se permiten:

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

Los valores `true` y `false` tienen la relación siguiente:

```
C++  
  
!false == true  
!true == false
```

En la instrucción siguiente:

```
C++  
  
if (condexpr1) statement1;
```

Si `condexpr1` es `true`, `statement1` siempre se ejecuta; si `condexpr1` es `false`, `statement1` nunca se ejecuta.

Cuando se aplica un operador `++` de prefijo o de postfijo a una variable de tipo `bool`, la variable se establece en `true`.

Visual Studio 2017, versión 15.3 y posteriores: se quitó del lenguaje `operator++` para `bool` y ya no se admite.

El operador `--` de prefijo o de postfijo no se puede aplicar a una variable de este tipo.

El tipo `bool` participa en promociones enteras predeterminadas. Un valor R de tipo `bool` se puede convertir en un valor R de tipo `int`, con `false` como cero y `true` como uno. Como un tipo distinto, `bool` participa en la resolución de sobrecarga.

Consulte también

[Palabras clave](#)

[Tipos integrados](#)

false (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

La palabra clave es uno de los dos valores para una variable de tipo `bool` o una expresión condicional (una expresión condicional es ahora una expresión true Booleana `true`). Por ejemplo, si `i` es una variable de tipo `bool`, la instrucción `i = false;` asigna false `false` a `i`.

Ejemplo

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
```

Consulte también

[Palabras clave](#)

true (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Sintaxis

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

Comentarios

Esta palabra clave es uno de los dos valores para una variable de tipo `bool` o una expresión condicional (una expresión condicional es ahora una expresión booleana `true`). Si `i` es de tipo `bool`, la instrucción `i = true;` asigna `true` a `i`.

Ejemplo

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
```

Consulte también

[Palabras clave](#)

char, wchar_t, char8_t, char16_t, char32_t

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Los tipos `char`, `wchar_t`, `char8_t`, `char16_t` y `char32_t` se compilan en los tipos integrados que representan caracteres alfanuméricos, glifos no alfanuméricos y caracteres no imprimibles.

Sintaxis

C++

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

Comentarios

El tipo `char` era el tipo de carácter original de C y C++. El tipo `char` se puede usar para almacenar caracteres del juego de caracteres ASCII o de cualquiera de los juegos de caracteres ISO-8859, así como bytes individuales de caracteres multibyte, como Shift-JIS o la codificación UTF-8 del juego de caracteres Unicode. En el compilador de Microsoft, `char` es un tipo de 8 bits. Es un tipo distinto tanto de `signed char` como de `unsigned char`. De forma predeterminada, las variables de tipo `char` se promueven a `int` como si fueran de tipo `signed char`, a menos que se use la opción del compilador `/J`. En `/J`, se tratan como de tipo `unsigned char` y se promueven a `int` sin la extensión de signo.

El tipo `unsigned char` se usa a menudo para representar un *byte* que no corresponde a un tipo integrado en C++.

El tipo `wchar_t` es un tipo de carácter ancho definido por la implementación. En el compilador de Microsoft, representa un carácter ancho de 16 bits que se usa para almacenar Unicode codificado como UTF-16LE, que es el tipo de carácter nativo en los sistemas operativos Windows. Las versiones de carácter ancho de las funciones de la biblioteca Universal C Runtime (UCRT) usan `wchar_t` y sus tipos de puntero y de matriz como parámetros y valores devueltos, al igual que las versiones de carácter ancho de la API nativa de Windows.

Los tipos `char8_t`, `char16_t` y `char32_t` representan caracteres anchos de 8, 16 y 32 bits respectivamente (`char8_t` es nuevo en C++20 y requiere la opción del compilador

`/std:c++20` o `/std:c++latest`). El Unicode codificado como UTF-8 se puede almacenar en el tipo `char8_t`. Las cadenas de tipo `char8_t` y `char` se conocen como cadenas *estrechas*, incluso cuando se usan para codificar caracteres Unicode o multibyte. El Unicode codificado como UTF-16 se puede almacenar en el tipo `char16_t` y el Unicode codificado como UTF-32, en el tipo `char32_t`. Las cadenas de estos tipos y `wchar_t` se conocen como cadenas *anchas*, aunque el término a menudo hace referencia específicamente a cadenas de tipo `wchar_t`.

En la biblioteca estándar de C++, el tipo `basic_string` está especializado en cadenas anchas y estrechas. Use `std::string` cuando los caracteres sean de tipo `char`; `std::u8string` cuando los caracteres sean de tipo `char8_t`; `std::u16string` cuando los caracteres sean de tipo `char16_t`; `std::u32string` cuando los caracteres sean de tipo `char32_t`, y `std::wstring` cuando los caracteres sean de tipo `wchar_t`. Otros tipos que representan texto, incluidos `std::stringstream` y `std::cout`, disponen de especializaciones de cadenas anchas y estrechas.

__int8, __int16, __int32, __int64

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específico de Microsoft

Las características de Microsoft C/C++ admiten tipos enteros con tamaño. Se pueden declarar variables de entero de 8, 16, 32 o 64 bits mediante el especificador de tipo `__intN`, donde `N` es 8, 16, 32 o 64.

En el ejemplo siguiente se declara una variable para cada uno de estos tipos de enteros con tamaño:

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
```

Los tipos `__int8`, `__int16` e `__int32` son sinónimos de los tipos ANSI que tienen el mismo tamaño, y son útiles para escribir código portable que se comporta de forma idéntica en varias plataformas. El tipo de datos `__int8` es sinónimo del tipo `char`, `__int16` es sinónimo del tipo `short` y `__int32` es sinónimo del tipo `int`. El tipo `__int64` es sinónimo del tipo `long long`.

A efectos de compatibilidad con versiones anteriores, `__int8`, `__int16`, `__int32` y `__int64` son sinónimos de `int8`, `int16`, `int32` y `int64` a menos que se especifique la opción del compilador [/Za \(Deshabilitar extensiones de lenguaje\)](#).

Ejemplo

En el ejemplo siguiente se muestra que un parámetro `__intN` se promoverá a `int`:

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
```

Consulte también

[Palabras clave](#)

[Tipos integrados](#)

[Intervalos de tipo de datos](#)

__m64

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

El tipo de datos `__m64` está destinado para usarse con funciones intrínsecas MMX y 3DNow! y se define en `<xmmmintrin.h>`.

C++

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

Comentarios

No debe acceder a los campos `__m64` directamente. Puede, sin embargo, ver estos tipos en el depurador. Una variable de tipo `__m64` se asigna a los registros MM[0-7].

Las variables de tipo `_m64` se alinean automáticamente en límites de 8 bytes.

El tipo de datos `__m64` no se admite en procesadores x64. Las aplicaciones que usan `_m64` como parte de intrínsecos de MMX se deben reescribir para usar intrínsecos de SSE y SSE2.

FIN de Específicos de Microsoft

Consulte también

[Palabras clave](#)

[Tipos integrados](#)

[Intervalos de tipo de datos](#)

__m128

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Define el tipo de datos **__m128** para su uso con las instrucciones de Extensiones SIMD de streaming y Extensiones SIMD de streaming 2 intrínsecas en <xmmmintrin.h>.

C++

```
// data_types__m128.cpp
#include <xmmmintrin.h>
int main() {
    __m128 x;
}
```

Comentarios

No debe acceder a los campos **__m128** directamente. Puede, sin embargo, ver estos tipos en el depurador. Una variable de tipo **__m128** se asigna a los registros XMM[0-7].

Las variables de tipo **__m128** se alinean automáticamente en límites de 16 bytes.

El tipo de datos **__m128** no se admite en procesadores ARM.

FIN de Específicos de Microsoft

Consulte también

[Palabras clave](#)

[Tipos integrados](#)

[Intervalos de tipo de datos](#)

__m128d

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

El tipo de datos `__m128d`, para su uso con las instrucciones intrínsecas de Extensiones SIMD de transmisión por secuencias 2, se define en `<emmintrin.h>`.

C++

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

Comentarios

No debe acceder a los campos `__m128d` directamente. Puede, sin embargo, ver estos tipos en el depurador. Una variable de tipo `__m128` se asigna a los registros XMM[0-7].

Las variables de tipo `_m128d` se alinean automáticamente en límites de 16 bytes.

El tipo de datos `__m128d` no se admite en procesadores ARM.

FIN de Específicos de Microsoft

Consulte también

[Palabras clave](#)

[Tipos integrados](#)

[Intervalos de tipo de datos](#)

__m128i

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

El tipo de datos **__m128i**, para su uso con las instrucciones intrínsecas de Extensiones SIMD de streaming 2 (SSE2), se define en <emmintrin.h>.

C++

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

Comentarios

No debe acceder a los campos **__m128i** directamente. Puede, sin embargo, ver estos tipos en el depurador. Una variable de tipo **__m128i** se asigna a los registros XMM[0-7].

Las variables de tipo **__m128i** se alinean automáticamente en límites de 16 bytes.

ⓘ Nota

El uso de variables de tipo **__m128i** hará que el compilador genere la instrucción **movdqa** de SSE2. Esta instrucción no produce ningún error en procesadores Pentium III, pero provocará un error silencioso, con posibles efectos secundarios debidos a las traducciones de las instrucciones **movdqa** en procesadores Pentium III.

El tipo de datos **__m128i** no se admite en procesadores ARM.

FIN de Específicos de Microsoft

Consulte también

[Palabras clave](#)

[Tipos integrados](#)

[Intervalos de tipo de datos](#)

`__ptr32`, `__ptr64`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

`__ptr32` representa un puntero nativo en un sistema de 32 bits, mientras que `__ptr64` representa un puntero nativo en un sistema de 64 bits.

En el ejemplo siguiente se muestra cómo declarar cada uno de estos tipos de puntero:

C++

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

En un sistema de 32 bits, un puntero declarado con `__ptr64` se trunca en un puntero de 32 bits. En un sistema de 64 bits, un puntero declarado con `__ptr32` se convierte en un puntero de 64 bits.

ⓘ Nota

No se puede utilizar `__ptr32` o `__ptr64` al compilar con `/clr:pure`. De lo contrario, se generará el error de compilador C2472. Las opciones del compilador `/clr:pure` y `/clr:safe` han quedado en desuso en Visual Studio 2015 y no se admiten en Visual Studio 2017.

Por compatibilidad con versiones anteriores, `_ptr32` y `_ptr64` son sinónimos de `__ptr32` y `__ptr64` a menos que se especifique la opción del compilador [/Za \(Deshabilitar extensiones de lenguaje\)](#).

Ejemplo

En el ejemplo siguiente se muestra cómo declarar y asignar punteros con las palabras clave `__ptr32` y `__ptr64`.

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
```

FIN de Específicos de Microsoft

Consulte también

[Tipos integrados](#)

Límites numéricos (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Los dos archivos de inclusión estándar, `<limits.h>` y `<float.h>`, definen los límites numéricos, o los valores mínimo y máximo que puede contener una variable de un tipo determinado. Se garantiza que estos valores mínimos y máximos son portátiles para cualquier compilador de C++ que use la misma representación de datos que ANSI C. El archivo de inclusión `<limits.h>` define los límites numéricos [para los tipos enteros](#) y `<float.h>` define los límites numéricos [para los tipos flotantes](#).

Consulte también

[Conceptos básicos](#)

Límites de enteros

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específico de Microsoft

Los límites de los tipos enteros se muestran en la tabla siguiente. Las macros de preprocesador para estos límites también se definen cuando se incluyen los <climits> del archivo de encabezado estándar.

Límites en constantes de enteros

| Constante | Significado | Valor |
|-------------------------|--|---------------------------------|
| <code>CHAR_BIT</code> | Número de bits de la variable menor que no es un campo de bits. | 8 |
| <code>SCHAR_MIN</code> | Valor mínimo de una variable de tipo <code>signed char</code> . | -128 |
| <code>SCHAR_MAX</code> | Valor máximo de una variable de tipo <code>signed char</code> . | 127 |
| <code>UCHAR_MAX</code> | Valor máximo de una variable de tipo <code>unsigned char</code> . | 255 (0xff) |
| <code>CHAR_MIN</code> | Valor mínimo de una variable de tipo <code>char</code> . | -128; 0 si se usa la opción /J |
| <code>CHAR_MAX</code> | Valor máximo de una variable de tipo <code>char</code> . | 127; 255 si se usa la opción /J |
| <code>MB_LEN_MAX</code> | Número máximo de bytes de una constante de varios caracteres. | 5 |
| <code>SHRT_MIN</code> | Valor mínimo de una variable de tipo <code>short</code> . | -32768 |
| <code>SHRT_MAX</code> | Valor máximo de una variable de tipo <code>short</code> . | 32767 |
| <code>USHRT_MAX</code> | Valor máximo de una variable de tipo <code>unsigned short</code> . | 65535 (0xffff) |
| <code>INT_MIN</code> | Valor mínimo de una variable de tipo <code>int</code> . | -2147483648 |
| <code>INT_MAX</code> | Valor máximo de una variable de tipo <code>int</code> . | 2147483647 |
| <code>UINT_MAX</code> | Valor máximo de una variable de tipo <code>unsigned int</code> . | 4294967295 (0xffffffff) |
| <code>LONG_MIN</code> | Valor mínimo de una variable de tipo <code>long</code> . | -2147483648 |

| Constante | Significado | Valor |
|-------------------------|--|--|
| <code>LONG_MAX</code> | Valor máximo de una variable de tipo <code>long</code> . | 2147483647 |
| <code>ULONG_MAX</code> | Valor máximo de una variable de tipo <code>unsigned long</code> . | 4294967295 (0xffffffff) |
| <code>LLONG_MIN</code> | Valor mínimo de una variable de tipo <code>long long</code> | -9223372036854775808 |
| <code>LLONG_MAX</code> | Valor máximo de una variable de tipo <code>long long</code> | 9223372036854775807 |
| <code>ULLONG_MAX</code> | Valor máximo de una variable de tipo <code>unsigned long long</code> | 18446744073709551615 (0xffffffffffffffffffff) |

Si un valor supera la representación de entero mayor, el compilador de Microsoft genera un error.

Consulte también

[Límites flotantes](#)

Límites flotantes

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

En la tabla siguiente se hace una lista de los límites de los valores de las constantes de punto flotante. Estos límites también se definen en el archivo de encabezado estándar `<float.h>`.

Límites en constantes de punto flotante

| Constante | Significado | Valor |
|------------------------------|--|-------------------------|
| <code>FLT_DIG</code> | Número de dígitos, q, tal que un número de punto flotante con q dígitos decimales se puede redondear en una representación de punto flotante y restablecer sin pérdida de precisión. | 6 |
| <code>DBL_DIG</code> | | 15 |
| <code>LDBL_DIG</code> | | 15 |
| <code>FLT_EPSILON</code> | Número positivo menor x, tal que x + 1,0 no es igual a 1,0. | 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> | Número de dígitos en la base especificada por <code>FLT_RADIX</code> en el significado de punto flotante. | 24 |
| <code>DBL_MANT_DIG</code> | | 53 |
| <code>LDBL_MANT_DIG</code> | La base es 2; por consiguiente, estos valores especifican los bits. | 53 |
| <code>FLT_MAX</code> | Número de punto flotante máximo que se puede representar. | 3.402823466e+38F |
| <code>DBL_MAX</code> | | 1.7976931348623158e+308 |
| <code>LDBL_MAX</code> | | 1.7976931348623158e+308 |
| <code>FLT_MAX_10_EXP</code> | Entero máximo tal que 10 elevado a dicho número es un número de punto flotante que se puede representar. | 38 |
| <code>DBL_MAX_10_EXP</code> | | 308 |
| <code>LDBL_MAX_10_EXP</code> | | 308 |
| <code>FLT_MAX_EXP</code> | Entero máximo tal que <code>FLT_RADIX</code> elevado a dicho número es un número de punto flotante que se puede representar. | 128 |
| <code>DBL_MAX_EXP</code> | | 1024 |
| <code>LDBL_MAX_EXP</code> | | 1024 |
| <code>FLT_MIN</code> | Valor positivo mínimo. | 1.175494351e-38F |
| <code>DBL_MIN</code> | | 2.2250738585072014e-308 |
| <code>LDBL_MIN</code> | | 2.2250738585072014e-308 |

| Constante | Significado | Valor |
|------------------------------|---|-------------|
| <code>FLT_MIN_10_EXP</code> | Entero negativo mínimo tal que 10 elevado a dicho número es un número de punto flotante que se puede representar. | -37 |
| <code>DBL_MIN_10_EXP</code> | | -307 |
| <code>LDBL_MIN_10_EXP</code> | | -307 |
| <code>FLT_MIN_EXP</code> | Entero negativo mínimo tal que <code>FLT_RADIX</code> elevado a dicho número es un número de punto flotante que se puede representar. | -125 |
| <code>DBL_MIN_EXP</code> | | -1021 |
| <code>LDBL_MIN_EXP</code> | | -1021 |
| <code>FLT_NORMALIZE</code> | | 0 |
| <code>FLT_RADIX</code> | Base de representación de exponente. | 2 |
| <code>_DBL_RADIX</code> | | 2 |
| <code>_LDBL_RADIX</code> | | 2 |
| <code>FLT_ROUNDS</code> | Modo de redondeo para la adición de punto flotante. | 1 (próximo) |
| <code>_DBL_ROUNDS</code> | | 1 (próximo) |
| <code>_LDBL_ROUNDS</code> | | 1 (próximo) |

ⓘ Nota

La información de la tabla puede ser diferente en versiones futuras del producto.

FIN de Específicos de Microsoft

Consulte también

[Límites de enteros](#)

Declaraciones y definiciones (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 5 minutos

Un programa de C++ consta de varias entidades, como variables, funciones, tipos y espacios de nombres. Cada una de estas entidades debe *declararse* para que puedan usarse. Una declaración especifica un nombre único para la entidad, junto con información sobre su tipo y otras características. En C++, el punto en el que se declara un nombre es el punto en el que se vuelve visible para el compilador. No puede referirse a una función o clase que se declara en un punto posterior en la unidad de compilación. Las variables se deben declarar lo más cerca posible antes del punto en el que se usan.

El siguiente ejemplo muestra algunas declaraciones:

```
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{/*...*/};
}
```

En la línea 5, se declara la función `main`. En la línea 7, una variable `const` denominada `pi` se declara e *inicializa*. En la línea 8, un entero `i` se declara e inicializa con el valor generado por la función `f`. El nombre `f` es visible para el compilador debido a la *declaración de desviar* en la línea 3.

En la línea 9, se declara una variable denominada `obj` de tipo `C`. Sin embargo, esta declaración genera un error porque `C` no se declara sino hasta más adelante en el

programa y no se declara directamente. Para corregir el error, puede mover toda la *definición* de `c` antes de `main` o bien agregar una declaración directa para ella. Este comportamiento es diferente de otros lenguajes, como C#. En esos lenguajes, se pueden usar funciones y clases antes de su punto de declaración en un archivo de origen.

En la línea 10, se declara una variable denominada `str` de tipo `std::string`. El nombre `std::string` es visible porque se introduce en el `string` [archivo de encabezado](#), que se combina en el archivo de origen en la línea 1. `std` es el espacio de nombres en el que se declara la clase `string`.

En la línea 11, se produce un error porque el nombre `j` no se ha declarado. Una declaración debe proporcionar un tipo, a diferencia de otros lenguajes, como JavaScript. En la línea 12, se usa la palabra clave `auto`, que indica al compilador que infiere el tipo de `k` en función del valor con el que se inicializa. En este caso, el compilador elige `int` para el tipo.

Ámbito de la declaración

El nombre introducido por una declaración es válido dentro del *ámbito* en el que se produce la declaración. En el ejemplo anterior, las variables declaradas dentro de la función `main` son *variables locales*. Podría declarar otra variable denominada `i` además de la principal, en el *ámbito global*, y sería una entidad independiente. Sin embargo, esta duplicación de nombres puede provocar confusión y errores del programador, y debe evitarse. En la línea 21, la clase `C` se declara en el ámbito del espacio de nombres `N`. El uso de espacios de nombres ayuda a evitar *colisiones de nombres*. La mayoría de los nombres de la biblioteca estándar de C++ se declara en el espacio de nombres `std`. Para obtener más información sobre la interacción entre las reglas de ámbito y las declaraciones, consulte [Ámbito](#).

Definiciones

Algunas entidades, incluidas las funciones, las clases, las enumeraciones y las variables constantes, deben definirse así como declararse. Una *definición* proporciona al compilador toda la información que necesita para generar código de máquina cuando la entidad se usa más adelante en el programa. En el ejemplo anterior, la línea 3 contiene una declaración para la función `f`, pero la *definición* de la función se proporciona en las líneas 15 a 18. En la línea 21, la clase `C` se declara y define (aunque tal como se define, la clase no hace nada). Se debe definir una variable constante; en otras palabras, se le debe asignar un valor, en la misma instrucción en la que se declara.

Una declaración de un tipo integrado, como `int`, constituye automáticamente una definición porque el compilador sabe cuánto espacio asignar para ella.

El ejemplo siguiente muestra declaraciones que también son definiciones:

```
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;  
};
```

Estas son algunas declaraciones que no son definiciones:

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

Definiciones de tipo e instrucciones "using"

En versiones anteriores de C++, la palabra clave `typedef` se usa para declarar un nuevo nombre que es un *alias* para otro nombre. Por ejemplo, el tipo `std::string` es otro nombre para `std::basic_string<char>`. Debe ser obvio por qué los programadores usan el nombre de definición de tipo y no el nombre real. En C++ moderno, se prefiere la palabra clave `using` a `typedef`, pero la idea es la misma: se declara un nuevo nombre para una entidad, que ya está declarada y definida.

Miembros de clases estáticas

Los miembros de datos de clase estática son variables discretas compartidas por todos los objetos de la clase. Dado que se comparten, deben definirse e inicializarse fuera de la definición de clase. Para más información, consulte [Clases](#).

declaraciones externas

Un programa de C++ podría contener más de una unidad de compilación. Para declarar una entidad que se define en una unidad de compilación independiente, use la palabra clave `extern`. La información de la declaración es suficiente para el compilador. Sin embargo, si la definición de la entidad no se encuentra en el paso de vinculación, entonces el enlazador generará un error.

En esta sección

[Clases de almacenamiento](#)

`const`

`constexpr`

`extern`

[Inicializadores](#)

[Alias y definiciones de tipo](#)

[using declaración](#)

`volatile`

`decltype`

[Atributos en C++](#)

Consulte también

[Conceptos básicos](#)

Clases de almacenamiento

Artículo • 03/03/2023 • Tiempo de lectura: 8 minutos

Una *clase de almacenamiento* en el contexto de declaraciones de variable de C++ es un especificador de tipo que rige la ubicación de memoria, la vinculación y la duración de objetos. Un objeto determinado puede tener solo una clase de almacenamiento. Las variables definidas dentro de un bloque tienen almacenamiento automático, a menos que se especifique lo contrario con los especificadores `extern`, `static` o `thread_local`. Las variables y objetos automáticos no tienen ninguna vinculación; no son visibles para código fuera del bloque. La memoria se les asigna automáticamente cuando la ejecución entra en el bloque y se desasigna cuando se sale del bloque.

Notas

- La palabra clave `mutable` puede considerarse un especificador de clase de almacenamiento. Sin embargo, solo está disponible en la lista de miembros de una definición de clase.
- **Visual Studio 2010 y versiones posteriores:** la palabra clave `auto` ya no es un especificador de clase de almacenamiento de C++ y la palabra clave `register` está en desuso. **Visual Studio 2017, versión 15.7 y posteriores:** (disponible en modo `/std:c++17` y posteriores): la palabra clave `register` se quita del lenguaje C++. Su uso provoca un mensaje de diagnóstico:

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

La palabra clave `static` puede usarse para declarar variables y funciones en el ámbito global, el ámbito de espacio de nombres y el ámbito de clase. También se pueden declarar variables estáticas en el ámbito local.

Duración estática significa que el objeto o la variable se asignan cuando se inicia el programa y se desasignan cuando finaliza el programa. Vinculación externa significa que el nombre de la variable puede verse desde fuera del archivo en el que se declara la

variable. A la inversa, la vinculación interna significa que el nombre no es visible fuera del archivo en el que se declara la variable. De forma predeterminada, una variable o un objeto que se defina en el espacio de nombres global tiene duración estática y vinculación externa. La palabra clave `static` se puede usar en las situaciones siguientes.

1. Cuando se declara una variable o función en el ámbito de archivo (ámbito global o de espacio de nombres), la palabra clave `static` especifica que la variable o función tienen vinculación interna. Cuando se declara una variable, la variable tiene duración estática y el compilador la inicializa como 0, a menos que se especifique otro valor.
2. Cuando se declara una variable en una función, la palabra clave `static` especifica que la variable mantiene su estado entre las llamadas a esa función.
3. Al declarar un miembro de datos en una declaración de clase, la palabra clave `static` especifica que todas las instancias de la clase comparten una copia del miembro. Un miembro de datos `static` se debe definir en el ámbito de archivo. Un miembro de datos entero que se declara como `const static` puede tener un inicializador.
4. Cuando se declara una función miembro en una declaración de clase, la palabra clave `static` especifica que todas las instancias de la clase comparten la función. Una función miembro `static` no puede acceder a un miembro de instancia porque la función no tiene un puntero `this` implícito. Para acceder a un miembro de instancia, declare la función con un parámetro que sea una referencia o un puntero de instancia.
5. No se pueden declarar los miembros de `union` como `static`. Sin embargo, una variable anónima declarada globalmente `union` se debe declarar `static` de manera explícita.

En este ejemplo, se muestra cómo una variable declarada `static` en una función conserva su estado entre las llamadas a esa función.

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
```

En este ejemplo, se muestra el uso de `static` en una clase.

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
3
3
```

En el ejemplo siguiente, se muestra una variable local declarada `static` en una función miembro. La variable `static` está disponible para el programa completo; todas las instancias del tipo comparten la misma copia de la variable `static`.

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
```

A partir de C++11, se garantiza que la inicialización de una variable local `static` sea segura para subprocessos. A veces, esta característica se denomina *estática mágica*. Sin embargo, en una aplicación con subprocesamiento múltiple todas las asignaciones

posteriores deben estar sincronizadas. La característica de inicialización estática segura para subprocessos se puede deshabilitar mediante la marca `/Zc:threadSafeInit-` para evitar depender de CRT.

extern

Los objetos y variables declarados como `extern` declaran un objeto definido en otra unidad de traducción o en un ámbito envolvente como si tuvieran vinculación externa. Para más información, consulte [extern](#) y [Unidades de traducción y vinculación](#).

thread_local (C++11)

Una variable declarada con el especificador `thread_local` solo es accesible en el subprocesso en el que se crea. La variable se crea cuando se crea el subprocesso y se destruye cuando se destruye el subprocesso. Cada subprocesso tiene su propia copia de la variable. En Windows, `thread_local` es funcionalmente equivalente al atributo `_declspec(thread)` específico de Microsoft.

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;
}
```

Aspectos a tener en cuenta sobre el especificador `thread_local`:

- Es posible que las variables locales de subprocessos inicializadas dinámicamente en archivos DLL no se inicialicen de manera correcta en todos los subprocessos de llamada. Para obtener más información, vea [thread](#).
- El especificado `thread_local` puede combinarse con `static` o `extern`.

- Solo puede aplicar `thread_local` a declaraciones y definiciones de datos; `thread_local` no se puede usar en declaraciones o definiciones de función.
- Solo puede especificar `thread_local` en elementos de datos con duración de almacenamiento estática, que incluye objetos de datos globales (`static` y `extern`), objetos estáticos locales y miembros de datos estáticos de clases. Cualquier variable local declarada `thread_local` es implícitamente estática si no se proporciona ninguna otra clase de almacenamiento; es decir, en el ámbito de bloque `thread_local` es equivalente a `thread_local static`.
- Debe especificar `thread_local` para la declaración y la definición de un objeto local de subproceso, ya sea que la declaración y la definición se realicen en el mismo archivo o en archivos distintos.
- No se recomienda usar variables `thread_local` con `std::launch::async`. Para más información, consulte [funciones <future>](#).

En Windows, `thread_local` es funcionalmente equivalente a `_declspec(thread)`, salvo en que `*__declspec(thread)*` se puede aplicar a una definición de tipo y es válido en código de C. Siempre que sea posible, use `thread_local` porque forma parte del estándar de C++ y, por tanto, es más portable.

registro

Visual Studio 2017, versión 15.3 y posteriores (disponible en modo `/std:c++17` y posteriores): la palabra clave `register` ya no es una clase de almacenamiento compatible. Su uso provoca un diagnóstico. La palabra clave todavía está reservada en el estándar para su uso futuro.

C++

```
register int val; // warning C5033: 'register' is no longer a supported
storage class
```

Ejemplo: inicialización automática frente a estática

Una variable o un objeto automático local se inicializa cada vez que el flujo de control alcanza su definición. Una variable o un objeto estático local se inicializa la primera vez que el flujo de control alcanza su definición.

Considere el ejemplo siguiente, que define una clase que registra la inicialización y la destrucción de objetos y, a continuación, define tres objetos, I1, I2 e I3:

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
```

En este ejemplo, se muestra cómo y cuándo se inicializan y destruyen los objetos `I1`, `I2` y `I3`.

Hay varios puntos que observar sobre el programa:

- En primer lugar, `I1` e `I2` se destruyen automáticamente cuando el flujo de control sale del bloque en el que están definidos.
- En segundo lugar, en C++, no es necesario declarar objetos o variables al principio de un bloque. Además, estos objetos se inicializan solo cuando el flujo de control alcanza sus definiciones. (`I2` y `I3` son ejemplos de estas definiciones). La salida muestra exactamente cuándo se inicializan.
- Por último, las variables locales estáticas como `I3` conservan sus valores mientras se ejecuta el programa, pero se destruyen en cuanto este finaliza.

Consulte también

[Declaraciones y definiciones](#)

auto (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 7 minutos

Deduce el tipo de una variable declarada a partir de su expresión de inicialización.

ⓘ Nota

El estándar C++ define un significado original y otro revisado de esta palabra clave. Antes de Visual Studio 2010, la palabra clave `auto` declara una variable en la clase de almacenamiento *automática*; es decir, una variable que tiene una duración local. A partir de Visual Studio 2010, la palabra clave `auto` declara un variable cuyo tipo se deduce de la expresión de inicialización de su declaración. La opción del compilador `/Zc:auto[-]` controla el significado de la palabra clave `auto`.

Sintaxis

```
auto declaratorinitializer;
```

```
[](auto param1, auto param2) {};
```

Comentarios

La palabra clave `auto` indica al compilador que use la expresión de inicialización de una variable declarada, o el parámetro de expresión lambda, para deducir su tipo.

Se recomienda usar la palabra clave `auto` en la mayoría de las situaciones, a menos que quiera realmente una conversión, porque proporciona estas ventajas:

- **Robustez:** Si se cambia el tipo de la expresión, incluido cuando se cambia un tipo de valor devuelto de función, solo funciona.
- **Rendimiento:** Está garantizado que no hay ninguna conversión.
- **Facilidad de uso:** no es necesario preocuparse por las dificultades y los errores tipográficos al escribir los nombres de los tipos.
- **Eficacia:** la codificación puede ser más eficaz.

Casos de conversión en los que puede que no quiera usar `auto`:

- Quiere un tipo específico y nada más hará.
- En los tipos auxiliares de plantillas de expresión, por ejemplo, `(valarray+valarray)`.

Use la palabra clave `auto` en lugar de un tipo para declarar una variable, y especifique una expresión de inicialización. Además, puede modificar la palabra clave `auto` mediante especificadores y declaradores como `const`, `volatile`, puntero (`*`), referencia (`&`) y referencia a rvalue (`&&`). El compilador evalúa la expresión de inicialización y emplea esa información para deducir el tipo de la variable.

La expresión de inicialización `auto` puede tener varios formatos:

- Sintaxis de inicialización universal, como `auto a { 42 };`.
- Sintaxis de asignación, como `auto b = 0;`.
- Sintaxis de asignación universal, que combina los dos formatos anteriores, como `auto c = { 3.14159 };`.
- Inicialización directa, o sintaxis de estilo constructor, como `auto d(1.41421f);`.

Para obtener más información, vea [Inicializadores](#) y los ejemplos de código más adelante en este documento.

Cuando `auto` se usa para declarar el parámetro de bucle en una instrucción `for` basada en intervalos, usa una sintaxis de inicialización diferente, por ejemplo, `for (auto& i : iterable) do_action(i);`. Para obtener más información, vea [Instrucción for basada en intervalos \(C++\)](#).

La `auto` palabra clave es un marcador de posición para un tipo, pero no es un tipo. Por lo tanto, la `auto` palabra clave no se puede usar en conversiones o operadores como `sizeof` y (para C++/CLI). [typeid](#)

Utilidad

La palabra clave `auto` es una manera sencilla de declarar una variable que tiene un tipo complicado. Por ejemplo, se puede usar `auto` para declarar una variable en la que la expresión de inicialización implica plantillas, punteros a funciones o punteros a miembros.

También se puede usar `auto` para declarar e inicializar una variable en una expresión lambda. No puede declarar el tipo de la variable porque solo el compilador conoce el tipo de una expresión lambda. Para obtener más información, vea [Ejemplos de expresiones lambda](#).

Tipos de valor devuelto finales

Se puede usar `auto`, junto con el especificador de tipo `decltype`, como ayuda para escribir bibliotecas de plantillas. Use `auto` y `decltype` para declarar una plantilla de función cuyo tipo de valor devuelto dependa de los tipos de sus argumentos de plantilla. O bien, use `auto` y `decltype` para declarar una plantilla de función que encapsula una llamada a otra función y, a continuación, devuelva el tipo de valor devuelto de esa otra función. Para más información, consulte [decltype](#).

Referencias y calificadores cv

Usar `auto` referencias quitadas, `const` calificadores y `volatile` calificadores. Considere el ejemplo siguiente:

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;
}
```

En el ejemplo anterior, `myAuto` es `int` no una `int` referencia, por lo que la salida es `11 11`, no `11 12` como sería el caso si el calificador de referencia no hubiera sido eliminado por `auto`.

Deducción de tipos con inicializadores entre llaves (C++14)

En el ejemplo de código siguiente se muestra cómo inicializar una variable `auto` con llaves. Observe la diferencia entre B y C y entre A y E.

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;
}
```

Restricciones y mensajes de error

En la tabla siguiente se enumeran las restricciones de uso de la palabra clave `auto` y el mensaje de error de diagnóstico correspondiente que el compilador emite.

| Número de error | Descripción |
|-----------------|---|
| C3530 | La <code>auto</code> palabra clave no se puede combinar con ningún otro especificador de tipo. |
| C3531 | Un símbolo que se declara con la palabra clave <code>auto</code> debe tener un inicializador. |
| C3532 | Ha usado incorrectamente la palabra clave <code>auto</code> para declarar un tipo. Por ejemplo, declaró un tipo de valor devuelto de método o una matriz. |
| C3533, C3539 | Un parámetro o argumento de plantilla no se puede declarar con la <code>auto</code> palabra clave . |
| C3535 | No se puede declarar un parámetro de método o plantilla con la <code>auto</code> palabra clave . |
| C3536 | No se puede usar un símbolo antes de que se inicialice. En la práctica, significa que una variable no se puede usar para inicializarse. |
| C3537 | No se puede convertir a un tipo declarado con la <code>auto</code> palabra clave . |

| Número | Descripción de error |
|--------------|---|
| C3538 | Todos los símbolos de una lista de declaradores que se declara con la palabra clave <code>auto</code> deben resolverse en el mismo tipo. Para obtener más información, vea Declaraciones y definiciones . |
| C3540, C3541 | Los operadores <code>sizeof</code> y <code>typeid</code> no se pueden aplicar a un símbolo declarado con la palabra clave <code>auto</code> . |

Ejemplos

Estos fragmentos de código muestran algunas de las formas en que se puede usar la palabra clave `auto`.

Las declaraciones siguientes son equivalentes. En la primera instrucción, la variable `j` se declara para que sea de tipo `int`. En la segunda instrucción, se deduce que la variable `k` es de tipo `int` porque la expresión de inicialización (0) es un entero.

C++

```
int j = 0; // Variable j is explicitly type int.
auto k = 0; // Variable k is implicitly type int because 0 is an integer.
```

Las declaraciones siguientes son equivalentes, pero la segunda declaración es más sencilla que la primera. Una de las razones de más peso para usar la palabra clave `auto` es la sencillez.

C++

```
map<int,list<string>>::iterator i = m.begin();
auto i = m.begin();
```

En el fragmento de código siguiente se declara el tipo de las variables `iter` y `elem` cuando se inician los bucles `for` y range `for`.

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
    { /* ... */ }
}

```

En el fragmento de código siguiente se usa el operador `new` y la declaración de puntero para declarar punteros.

C++

```

double x = 12.34;
auto *y = new auto(x), **z = new auto(&x);

```

En el fragmento de código siguiente se declaran varios símbolos en cada instrucción de declaración. Observe que todos los símbolos de cada instrucción se resuelven en el mismo tipo.

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.

```

En este fragmento de código se utiliza el operador condicional (`:?`) para declarar la variable `x` como un entero que tiene un valor de 200:

C++

```

int v1 = 100, v2 = 200;
auto x = v1 > v2 ? v1 : v2;

```

En el fragmento de código siguiente se inicializa la variable `x` en el tipo `int`, la variable `y` en una referencia al tipo `const int` y la variable `fp` en un puntero a una función que devuelve el tipo `int`.

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;
    //...
}
```

Consulte también

[Palabras clave](#)

[/Zc:auto \(Deducir tipo de variable\)](#)

[sizeof operador](#)

[typeid](#)

[operator new](#)

[Declaraciones y definiciones](#)

[Ejemplos de expresiones lambda](#)

[Inicializadores](#)

[decltype](#)

const (C++)

Artículo • 14/03/2023 • Tiempo de lectura: 4 minutos

Cuando se modifica una declaración de datos, la palabra clave **const** especifica que el objeto o la variable no se puede modificar.

Sintaxis

```
declarator:  
    ptr-declarator  
    noptr-declarator parameters-and-qualifiers trailing-return-type  
  
ptr-declarator:  
    noptr-declarator  
    ptr-operator ptr-declarator  
  
noptr-declarator:  
    declarator-id attribute-specifier-seqopt  
    noptr-declarator parameters-and-qualifiers  
    noptr-declarator [ constant-expressionopt] attribute-specifier-seqopt  
    ( ptr-declarator )  
  
parameters-and-qualifiers:  
    ( parameter-declaration-clause ) cv-qualifier-seqopt  
    ref-qualifieropt noexcept-specifieropt attribute-specifier-seqopt  
  
trailing-return-type:  
    -> type-id  
  
ptr-operator:  
    * attribute-specifier-seqopt cv-qualifier-seqopt  
    & attribute-specifier-seqopt  
    && attribute-specifier-seqopt  
    nested-name-specifier * attribute-specifier-seqopt cv-qualifier-seqopt  
  
cv-qualifier-seq:  
    cv-qualifier cv-qualifier-seqopt  
  
cv-qualifier:  
    const  
    volatile  
  
ref-qualifier:  
    &  
    &&
```

declarator-id:

 ... opt *id-expression*

Valores **const**

La palabra clave **const** especifica que el valor de una variable es constante e indica al compilador que evite que el programador lo modifique.

C++

```
// constant_values1.cpp
int main() {
    const int i = 5;
    i = 10;    // C3892
    i++;      // C2105
}
```

En C++, se puede usar la palabra clave **const** en lugar de la directiva de preprocesador **#define** para definir valores constantes. Los valores definidos con **const** están sujetos a la comprobación de tipos y se pueden usar en lugar de expresiones constantes. En C++, puede especificar el tamaño de una matriz con una variable **const** de la forma siguiente:

C++

```
// constant_values2.cpp
// compile with: /c
const int maxarray = 255;
char store_char[maxarray]; // allowed in C++; not allowed in C
```

En C, los valores constantes tienen la vinculación externa como valor predeterminado, por lo que solo pueden aparecer en los archivos de código fuente. En C++, los valores constantes tienen la vinculación interna como valor predeterminado, que permite que aparezcan en los archivos de encabezado.

La palabra clave **const** también se puede usar en las declaraciones de puntero.

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
}
```

Un puntero a una variable declarada como `const` solo se puede asignar a un puntero que también se declare como `const`.

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
}
```

Puede utilizar punteros a datos constantes como parámetros de función para evitar que la función modifique un parámetro pasado a través de un puntero.

Para los objetos que se declaran como `const`, solo se puede llamar a las funciones miembro constante. El compilador garantiza que el objeto constante nunca se modifique.

C++

```
birthday.getMonth(); // Okay
birthday.setMonth( 4 ); // Error
```

Se puede llamar a funciones miembro constante o no constante para un objeto que no es constante. Una función miembro también se puede sobrecargar mediante la palabra clave `const`; esta característica permite que se llame a una versión diferente de la función para los objetos constante y que no son constante.

Los constructores o destructores no se pueden declarar con la palabra clave `const`.

Funciones miembro `const`

Declarar una función miembro con la palabra clave `const` indica que la función es una función de "solo lectura" que no modifica el objeto para el que se llama. Una función

miembro constante no puede modificar los miembros de datos no estáticos ni llamar a funciones miembro que no sean constante. Para declarar una función miembro constante, coloque la palabra clave `const` después del paréntesis de cierre de la lista de argumentos. La palabra clave `const` se requiere tanto en la declaración como en la definición.

```
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  
}
```

Diferencias de `const` en C y C++

Al definir una `const` variable en un archivo de código fuente de C, haga lo siguiente:

```
C  
  
const int i = 2;
```

A continuación, esta variable se puede utilizar en otro módulo como sigue:

```
C  
  
//
```

```
extern const int i;
```

Pero para obtener el mismo comportamiento en C++, debe definir la `const` variable como:

C++

```
extern const int i = 2;
```

De forma similar a C, puede usar esta variable en otro módulo de la siguiente manera:

C++

```
extern const int i;
```

Si desea definir una `extern` variable en un archivo de código fuente de C++ para su uso en un archivo de código fuente de C, use:

C++

```
extern "C" const int x=10;
```

para evitar que el compilador de C++ elimine nombres.

Comentarios

Cuando va después de la lista de parámetros de una función miembro, la palabra clave `const` especifica que la función no modifica el objeto para el que se invoca.

Para más información sobre `const`, consulte los artículos siguientes:

- Punteros [const y volatile](#)
- [Calificadores de tipo \(referencia del lenguaje C\)](#)
- [volatile](#)
- [#define](#)

Consulte también

[Palabras clave](#)

constexpr (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 6 minutos

La palabra clave `constexpr` se introdujo en C++11 y se mejoró en C++14. Significa *Expresión constant*. Al igual que `const`, se puede aplicar a las variables: se produce un error del compilador si algún código intenta modificar el valor. A diferencia de `const`, `constexpr` también se puede aplicar a los constructores de clase y funciones. `constexpr` indica que el valor o el valor devuelto, es constant y, si es posible, se procesará en tiempo de compilación.

Un valor entero `constexpr` se puede usar donde se requiera un entero `const`, como es el caso de las declaraciones de matrices y los argumentos de plantillas. Y si un valor se procesa en tiempo de compilación en lugar de en tiempo de ejecución, ayuda a su programa a ejecutarse más rápidamente y a usar menos memoria.

Para limitar la complejidad de los cálculos constant en tiempo de compilación y su posible impacto en este tiempo, el estándar de C++14 requiere los tipos en expresiones constant para ser [tipos literales](#).

Sintaxis

```
constexpr literal-type identifier=constant-expression;  
constexpr literal-type identifier{constant-expression};  
constexpr literal-type identifier(params);  
constexpr ctor(params);
```

Parámetros

params

Uno o más parámetros, los cuales deben ser cada uno un tipo literal y deben ser en sí mismos una expresión constant.

Valor devuelto

Una variable o función `constexpr` debe devolver un [tipo literal](#).

Variables `constexpr`

La diferencia entre las variables `const` y `constexpr` es que la inicialización de una variable `const` se puede aplazar hasta el tiempo de ejecución. Una variable `constexpr` se debe inicializar en tiempo de compilación. Todas las variables `constexpr` son `const`.

- Una variable se puede declarar con `constexpr`, si tiene un tipo literal y está inicializada. Si la inicialización es realizada por medio de un constructor, el constructor debe declararse como `constexpr`.
- Se puede declarar una referencia como `constexpr` cuando se cumplen estas condiciones: el objeto al que se hace referencia se inicializa mediante una expresión constant y las conversiones implícitas invocadas durante la inicialización también son expresiones constant.
- Todas las declaraciones de una función o variable `constexpr` deben tener el especificador `constexpr`.

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
```

funcionesconstexpr

Una función `constexpr` es aquella cuyo valor devuelto se calcula durante el tiempo de compilación cuando el código usado lo requiere. El código de consumo requiere el valor de retorno en tiempo de compilación para inicializar una variable `constexpr` o proporcionar un argumento de plantilla que no sea de tipo. Cuando sus argumentos son valores `constexpr`, una función `constexpr` genera una constant en tiempo de compilación. Cuando se llama con argumentos que no sean `constexpr` o cuando su valor no se requiere en tiempo de compilación, genera un valor en tiempo de ejecución como una función normal. (Este doble comportamiento le evita tener que escribir tanto versiones `constexpr` como las que no sean `constexpr` de la misma función).

Una función `constexpr` o constructor es implícitamente `inline`.

Las reglas siguientes se aplican a las funciones `constexpr`:

- Una función `constexpr` debe aceptar y devolver únicamente [tipos literales](#).

- Una función `constexpr` puede ser recursiva.
- Seaforc++20, una `constexpr` función no puede ser `virtual` y un constructor no se puede definir como `constexpr` cuando la clase envolvente tiene ninguna clase base virtual. En C++20 y versiones posteriores, una `constexpr` función puede ser virtual. Visual Studio 2019, versión 16.10 y versiones posteriores, admiten `constexpr` funciones virtuales al especificar la `/std:c++20` opción del compilador o posterior.
- El cuerpo se puede definir como `= default` o `= delete`.
- El cuerpo no puede contener instrucciones `goto` ni bloques `try`.
- Una especialización explícita de una plantilla que no sea `constexpr` no se puede declarar como `constexpr`:
- Una especialización explícita de una plantilla `constexpr` no tiene que ser también `constexpr`:

Las reglas siguientes se aplican a las funciones `constexpr` de Visual Studio 2017 y versiones posteriores:

- Puede contener instrucciones `if` y `switch`, y todas las instrucciones de bucle, incluidas `for`, basadas en intervalos `for`, `while` y `do-while`.
- Puede contener declaraciones de variables locales, pero la variable debe inicializarse. Debe ser un tipo literal y no puede ser `static` ni local para subprocessos. La variable declarada localmente no es necesaria para ser `const` y puede mutar.
- No es necesario que una función `constexpr` que no sea miembro `static` sea implícitamente `const`.

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;
}
```

💡 Sugerencia

En el depurador de Visual Studio, al depurar una compilación de depuración no optimizada, puede indicar si una función `constexpr` se evalúa en tiempo de compilación colocando un punto de interrupción en su interior. Si se alcanza el punto de interrupción, significa que se llamó a la función en tiempo de ejecución. En caso contrario, significa que se llamó a la función en tiempo de compilación.

constexpr externo

La opción del compilador `/Zc:externConstexpr` hace que el compilador aplique la [vinculación externa](#) a las variables declaradas mediante `constexpr` externo. En versiones anteriores de Visual Studio, ya sea de forma predeterminada o cuando `/Zc:externConstexpr-` especifica, Visual Studio aplica la vinculación interna a las variables `constexpr` incluso cuando se usa la palabra clave `extern`. La opción `/Zc:externConstexpr` está disponible a partir de la actualización 15.6 de Visual Studio 2017 y está desactivada de forma predeterminada. La opción `/permissive-` no habilita `/Zc:externConstexpr`.

Ejemplo

En el siguiente ejemplo se muestran las variables `constexpr`, las funciones y un tipo definido por usuario. En la última instrucción en `main()`, la función miembro `constexpr GetValue()` es una llamada en tiempo de ejecución porque no es necesario conocer el valor en tiempo de compilación.

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;
}

```

Requisitos

Visual Studio 2015 o posterior.

Consulte también

[Declaraciones y definiciones](#)

[const](#)

extern (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 4 minutos

La palabra clave `extern` se puede aplicar a una declaración de variable, función o plantilla global. Especifica que el símbolo tiene *externuna vinculación*. Para obtener información general sobre la vinculación y por qué no se recomienda el uso de variables globales, consulte [Unidades de traducción y vinculación](#).

La palabra clave `extern` tiene cuatro significados en función del contexto:

- En una declaración de variable no global `const`, `extern` especifica que la variable o función se define en otra unidad de traducción. `extern` debe aplicarse en todos los archivos excepto en el que se define la variable.
- En una declaración de variable `const`, especifica que la variable tiene una vinculación externa. `extern` debe aplicarse a todas las declaraciones de todos los archivos. (Las variables globales `const` tienen vinculación interna de forma predeterminada).
- `extern "C"` especifica que la función se define en otro lugar y usa la convención de llamada del lenguaje C. El modificador `extern "C"` también se puede aplicar a varias declaraciones de función en un bloque.
- En una declaración de plantilla, `extern` especifica que ya se ha creado una instancia de la plantilla en otro lugar. `extern` indica al compilador que puede reutilizar la otra creación de instancias, en lugar de crear una nueva en la ubicación actual. Para obtener más información sobre este uso de `extern`, vea [Creación de instancias explícitas](#).

extern vinculación para los no globales `const`

Cuando el enlazador ve `extern` antes de una declaración de variable global, busca la definición en otra unidad de traducción. Las declaraciones de no variables `const` que están en el ámbito global son extern de forma predeterminada. Solo se aplica `extern` a las declaraciones que no proporcionan la definición.

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)
```

extern vinculación para los globales const

Una variable global `const` tiene vinculación interna de forma predeterminada. Si desea que la variable tenga vinculación externa, aplique la palabra clave `extern` a la definición y a todas las demás declaraciones de otros archivos:

C++

```
//fileA.cpp
extern const int i = 42; // extern const definition

//fileB.cpp
extern const int i; // declaration only. same as i in FileA
```

Vinculación de extern constexpr

En versiones 15.3 o anteriores de Visual Studio 2017, el compilador siempre proporcionaba una vinculación interna de variable de `constexpr` aunque la variable se marcase como `extern`. En la versión 15.5 de Visual Studio 2017 y posterior, un nuevo conmutador de compilador `/Zc:externConstexpr` permite un comportamiento correcto que cumple con los estándares. Esta opción acabará convirtiéndose en el conmutador predeterminado. La opción `/permissive-` no habilita `/Zc:externConstexpr`.

C++

```
extern constexpr int x = 10; //error LNK2005: "int const x" already defined
```

Si un archivo de encabezado contiene una variable declarada como `extern constexpr`, debe marcase como `__declspec(selectany)` para que tengan sus declaraciones duplicadas combinadas:

C++

```
extern constexpr __declspec(selectany) int x = 10;
```

Declaraciones de función `extern "C"` y `extern "C++"`

En C++, cuando se usa con una cadena, `extern` especifica el uso de las convenciones de vinculación de otro lenguaje para los declaradores. Las funciones y datos de C solo están accesibles si se declaran previamente con vinculación de C. Sin embargo, se deben definir en una unidad de traducción compilada por separado.

Microsoft C++ admite las cadenas `"C"` y `"C++"` en el campo de *string-literal*. Todos los archivos de inclusión estándar utilizan la sintaxis de `extern "C"` para permitir que las funciones de la biblioteca en tiempo de ejecución se usen en programas de C++.

Ejemplo

En el ejemplo siguiente se muestran cómo declarar los nombres que tienen vinculación de C:

```
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;
```

Si una función tiene más de una especificación de vinculación, debe estar de acuerdo. Es un error declarar funciones como que tienen vinculación de C y C++. Además, si en un programa aparecen dos declaraciones para una función (una con una especificación de vinculación y otra sin ella), la declaración con especificación de vinculación debe ser la primera. Cualquier declaración redundante de funciones que ya tengan especificación de vinculación recibe la vinculación especificada en la primera declaración. Por ejemplo:

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.
```

A partir de Visual Studio 2019, cuando `/permissive-` se especifica, el compilador comprueba que las declaraciones de parámetros de función `extern "C"` también coinciden. No se puede sobrecargar una función declarada como `extern "C"`. A partir de la versión 16.3 de Visual Studio 2019, puede invalidar esta comprobación mediante la opción `/Zc:externC-` del compilador después de la opción `/permissive-`.

Consulte también

[Palabras clave](#)

[Unidades de traducción y vinculación](#)

[extern Especificadores Storage-Class en C](#)

[Comportamiento de identificadores en C](#)

[Vinculación en C](#)

Inicializadores

Artículo • 03/03/2023 • Tiempo de lectura: 13 minutos

Un inicializador especifica el valor inicial de una variable. Se pueden inicializar variables en estos contextos:

- En la definición de una variable:

C++

```
int i = 3;  
Point p1{ 1, 2 };
```

- Como uno de los parámetros de una función:

C++

```
set_point(Point{ 5, 6 });
```

- Como el valor devuelto de una función:

C++

```
Point get_new_point(int x, int y) { return { x, y }; }  
Point get_new_point(int x, int y) { return Point{ x, y }; }
```

Los inicializadores pueden adoptar estos formatos:

- Una expresión (o una lista de expresiones separadas por comas) entre paréntesis:

C++

```
Point p1(1, 2);
```

- Un signo igual seguido de una expresión:

C++

```
string s = "hello";
```

- Una lista de inicializadores entre llaves. La lista puede estar vacía o puede consistir en un conjunto de listas, como en el ejemplo siguiente:

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 } });
}
```

Clases de inicialización

Hay varias clases de inicialización, que pueden producirse en distintos puntos de la ejecución de un programa. Las diferentes clases de inicialización no son mutuamente excluyentes; por ejemplo, la inicialización de la lista puede desencadenar la inicialización de un valor y en otras circunstancias, puede desencadenar la inicialización de agregado.

Inicialización cero

La inicialización cero es la configuración de una variable en un valor cero convertido implícitamente al tipo:

- Las variables numéricas se inicializan en 0 (o 0,0, 0,0000000000, etc.).
- Las variables char se inicializan en '\0'.
- Los punteros se inicializan en `nullptr`.
- Los miembros de matrices, clases POD, estructuras y uniones se inicializan en un valor cero.

La inicialización cero se realiza en distintos momentos:

- Al iniciarse el programa, para todas las variables con nombre que tienen duración estática. Estas variables se pueden volver a inicializar posteriormente.
- Durante la inicialización del valor, para los tipos escalares y de clase POD que se inicializan mediante llaves vacías.

- Para las matrices en las que solo se inicializa un subconjunto de sus miembros.

A continuación se muestran algunos ejemplos de inicialización cero:

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 0.00000000000000000000
    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'
}
```

Inicialización predeterminada

La inicialización predeterminada de clases, structs y uniones es la inicialización con un constructor predeterminado. Se puede llamar al constructor predeterminado sin una expresión de inicialización o con la palabra clave `new`:

C++

```
MyClass mc1;
MyClass* mc3 = new MyClass;
```

Si la clase, la estructura o la unión no tiene un constructor predeterminado, el compilador emite un error.

Las variables escalares se inicializan de forma predeterminada cuando se definen sin una expresión de inicialización. Tienen valores indeterminados.

C++

```
int i1;
float f;
char c;
```

Las matrices se inicializan de forma predeterminada cuando se definen sin una expresión de inicialización. Cuando una matriz se inicializa de forma predeterminada, sus miembros también se inicializan de forma predeterminada y tienen valores indeterminados, como en el ejemplo siguiente:

C++

```
int int_arr[3];
```

Si los miembros de la matriz no tienen un constructor predeterminado, el compilador emite un error.

Inicialización predeterminada de variables constantes

Las variables constantes se deben declarar junto con un inicializador. Si son tipos escalares, generan un error del compilador, y si son tipos de clase que tienen un constructor predeterminado, generan una advertencia:

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  
}
```

Inicialización predeterminada de variables estáticas

Las variables estáticas que se declaran sin un inicializador se inicializan en 0 (se convierten implícitamente al tipo):

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'}
}
```

Para obtener más información sobre la inicialización de objetos estáticos globales, consulte los [argumentos de la función principal y de la línea de comandos](#).

Inicialización de un valor

La inicialización de un valor se produce en los siguientes casos:

- un valor con nombre se inicializa con llaves vacías
- un objeto temporal anónimo se inicializa con paréntesis o llaves vacíos
- un objeto se inicializa con la palabra clave `new` y paréntesis o llaves vacíos

La inicialización de un valor hace lo siguiente:

- para las clases que tienen al menos un constructor público, se llama al constructor predeterminado
- para las clases que no son de unión y no tienen constructores declarados, el objeto se inicializa en cero y se llama al constructor predeterminado
- para las matrices, se inicializa el valor de cada elemento
- en todos los demás casos, la variable se inicializa en cero

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]{};
    int a{};        // value of a is 0
    double b{};     // value of b is 0.0000000000000000
```

Inicialización de copia

La inicialización de copia es la inicialización de un objeto mediante otro objeto. Se produce en los casos siguientes:

- se inicializa una variable mediante un signo igual
- se pasa un argumento a una función
- se devuelve un objeto de una función
- se produce o detecta una excepción
- se inicializa un miembro de datos no estático con un signo igual
- se inicializan los miembros class, struct y union con la inicialización de copia durante la inicialización de agregado Consulte [Inicialización de agregado](#) para obtener ejemplos.

En el código siguiente se muestran varios ejemplos de inicialización de copia:

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();
    }
}
```

La inicialización de copia no puede invocar constructores explícitos:

C++

```
vector<int> v = 10; // the constructor is explicit; compiler error C2440:  
// cannot 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
```

En algunos casos, si el constructor de copias de la clase se ha eliminado o es inaccesible, la inicialización de copia produce un error del compilador.

Inicialización directa

La inicialización directa es la inicialización con llaves o paréntesis (no vacíos). A diferencia de la inicialización de copia, puede invocar constructores explícitos. Se produce en los casos siguientes:

- una variable se inicializa con llaves o paréntesis no vacíos
- una variable se inicializa con la palabra clave `new` y llaves o paréntesis no vacíos
- una variable se inicializa con `static_cast`
- en un constructor, las clases base y los miembros no estáticos se inicializan con una lista de inicializadores
- en la copia de una variable capturada en una expresión lambda

En el código siguiente se muestran algunos ejemplos de inicialización directa:

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();
}

```

Inicialización de listas

La inicialización de lista se produce cuando se inicializa una variable con una lista de inicializadores entre llaves. Se pueden usar listas de inicializadores entre llaves en los siguientes casos:

- se inicializa una variable
- se inicializa una clase con la palabra clave `new`
- se devuelve un objeto de una función
- se pasa un argumento a una función
- uno de los argumentos de una inicialización directa
- en un inicializador de miembros de datos no estáticos
- en una lista de inicializadores del constructor

En el código siguiente se muestran algunos ejemplos de inicialización de lista:

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' } };
}

```

Inicialización de agregado

La inicialización de agregado es una forma de inicialización de lista para matrices o tipos de clase (a menudo structs o uniones) que tienen:

- ningún miembro privado o protegido
- ningún constructor proporcionado por el usuario, salvo para los constructores eliminados o establecidos como valor predeterminado explícitamente
- ninguna clase base
- ninguna función miembro virtual

ⓘ Nota

En Visual Studio 2015 y versiones anteriores, no se permite que un agregado tenga inicializadores de llave o igualdad para miembros no estáticos. Esta restricción se eliminó en el estándar C++14 y se implementó en Visual Studio 2017.

Los inicializadores de agregado constan de una lista de inicialización entre llaves, con o sin un signo de igualdad, como en el ejemplo siguiente:

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;
}

```

Debería aparecer la siguiente salida:

Output

```

agg1: c: 1
agg2: Z: 2
myArr1: 1 2 3 4
myArr3: 8 9 10 0 0

```

ⓘ Importante

Los miembros de la matriz que se declararon, pero no se inicializaron explícitamente durante la inicialización de agregado, se inicializan en cero, como en `myArr3` más arriba.

Inicializar uniones y structs

Si una unión no tiene un constructor, puede inicializarla con un valor único (o con otra instancia de una unión). El valor se utiliza para inicializar el primer campo no estático.

Esto es diferente de la inicialización de struct, donde el primer valor del inicializador se utiliza para inicializar el primer campo, el segundo valor para inicializar el segundo campo, y así sucesivamente. Compare la inicialización de uniones y structs en el ejemplo siguiente:

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'
}
```

Inicializar agregados que contienen agregados

Los tipos agregados pueden contener otros tipos agregados, como matrices de matrices, matrices de structs, etc. Estos tipos se inicializan con conjuntos anidados de llaves, por ejemplo:

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' } };
}

```

Inicialización de referencia

Las variables de tipo de referencia se deben inicializar con un objeto del tipo del que se deriva el tipo de referencia o con un objeto de un tipo que se pueda convertir al tipo que se deriva del tipo de referencia. Por ejemplo:

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
}

```

La única manera de inicializar una referencia con un objeto temporal consiste en inicializar un objeto temporal constante. Una vez inicializado, una variable de tipo de referencia siempre señala al mismo objeto; no se puede modificar para que señale a otro objeto.

Aunque la sintaxis puede ser igual, la inicialización de variables de tipo de referencia y la asignación a variables de tipo de referencia son semánticamente diferentes. En el ejemplo anterior, las asignaciones que modifican `iVar` y `lVar` son similares a las inicializaciones, pero tienen efectos diferentes. La inicialización especifica el objeto al que señala la variable de tipo de referencia; la asignación asigna al objeto al que se hace referencia a través de la referencia.

Dado que tanto pasar un argumento de tipo de referencia a una función como devolver un valor de tipo de referencia desde una función son inicializaciones, los argumentos

formales a una función se inicializan correctamente, así como las referencias devueltas.

Las variables de tipo de referencia solo se pueden declarar sin inicializadores en lo siguiente:

- Declaraciones de función (prototipos). Por ejemplo:

C++

```
int func( int& );
```

- Declaraciones de tipo de valor devuelto de función. Por ejemplo:

C++

```
int& func( int& );
```

- Declaración de un miembro de clase de tipo de referencia. Por ejemplo:

C++

```
class c {public:    int& i;};
```

- Declaración de una variable especificada explícitamente como `extern`. Por ejemplo:

C++

```
extern int& iVal;
```

Al inicializar una variable de tipo de referencia, el compilador usa el gráfico de decisión que se muestra en la figura siguiente para elegir entre crear una referencia a un objeto o crear un objeto temporal al que señala la referencia.

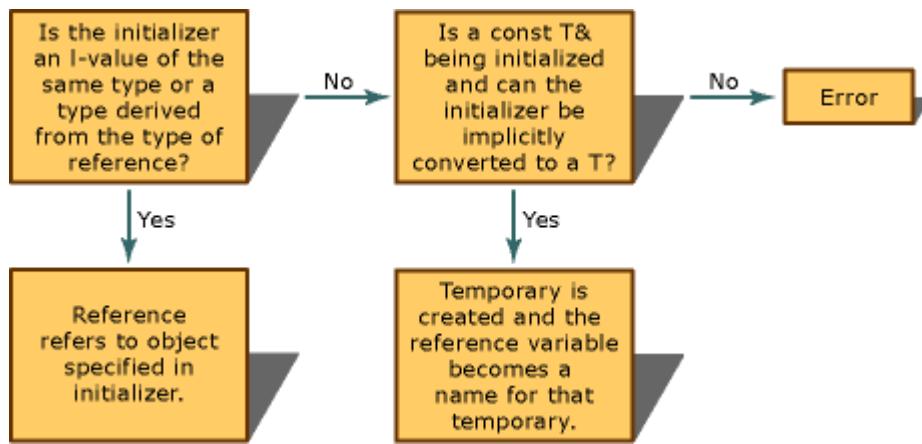


Gráfico de decisión de inicialización de tipos de referencia

Las referencias a tipos `volatile` (declarados como `volatile typename& identifier`) se pueden inicializar con objetos `volatile` del mismo tipo o con objetos que no se hayan declarado como `volatile`. En cambio, no pueden inicializarse con objetos `const` de ese tipo. De igual forma, las referencias a tipos `const` (declarados como `const typename& identifier`) se pueden inicializar con objetos `const` del mismo tipo (o con algo que tenga una conversión a ese tipo) o con objetos que no se hayan declarado como `const`. En cambio, no pueden inicializarse con objetos `volatile` de ese tipo.

Las referencias que no se califican con `const` o la palabra clave `volatile` solo se pueden inicializar con objetos que no se hayan declarado como `const` ni `volatile`.

Inicialización de variables externas

Las declaraciones de variables automáticas, estáticas y externas pueden contener inicializadores. Aun así, las declaraciones de variables externas solo pueden contener inicializadores si las variables no se declaran como `extern`.

Alias y definiciones de tipos (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 7 minutos

Se puede usar una *declaración de alias* para declarar un nombre que se usará como sinónimo de un tipo declarado previamente. (Este mecanismo también se conoce informalmente como *alias de tipo*). También se puede usar para crear una *plantilla de alias*, que puede resultar útil para los asignadores personalizados.

Sintaxis

C++

```
using identifier = type;
```

Comentarios

identifier

Nombre del alias.

type

Identificador de tipo para el que se creará un alias.

Un alias no presenta un tipo nuevo ni puede cambiar el significado de un nombre de tipo existente.

El formato más sencillo de alias es equivalente al mecanismo `typedef` de C++03:

C++

```
// C++11
using counter = long;

// C++03 equivalent:
// typedef long counter;
```

Ambos formatos permiten la creación de variables de tipo `counter`. Algo más útil sería un alias de tipo como este para `std::ios_base::fmtflags`:

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);
```

Los alias también funcionan con punteros de función, pero son mucho más legibles que la definición de tipo equivalente:

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;
```

Una limitación del mecanismo `typedef` es que no funciona con plantillas. Sin embargo, la sintaxis de alias de tipo de C++11 permite la creación de plantillas de alias:

C++

```
template<typename T> using ptr = T*;  
  
// the name 'ptr<T>' is now an alias for pointer to T  
ptr<int> ptr_int;
```

Ejemplo

En el ejemplo siguiente se muestra cómo utilizar una plantilla de alias con un asignador personalizado; en este caso, un tipo de vector entero. Puede sustituir cualquier tipo por `int` para crear un alias adecuado con el fin de ocultar las listas de parámetros complejos en el código funcional principal. El uso del asignador personalizado en todo el código puede mejorar la legibilidad y reducir el riesgo de introducir errores debidos a errores ortográficos.

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

Typedefs

Una declaración `typedef` introduce un nombre que, dentro de su ámbito, se convierte en un sinónimo del tipo especificado por la parte *type-declaration* de la declaración.

Puede usar declaraciones de definición de tipo para construir nombres más cortos o significativos para tipos ya definidos por el lenguaje o que ha declarado. Los nombres de `typedef` permiten encapsular detalles de la implementación que pueden cambiar.

A diferencia de las declaraciones `class`, `struct`, `union` y `enum`, las declaraciones `typedef` no introducen tipos nuevos, sino nombres nuevos para tipos ya creados.

Los nombres declarados con `typedef` ocupan el mismo espacio de nombres que otros identificadores (excepto las etiquetas de instrucciones). Por consiguiente, no pueden usar el mismo identificador que un nombre declarado previamente, excepto en una declaración de tipo de clase. Considere el ejemplo siguiente:

C++

```
// typedef_names1.cpp
// C2377 expected
typedef unsigned long UL;    // Declare a typedef name, UL.
int UL;                      // C2377: redefined.
```

Las reglas de nombres ocultos que pertenecen a otros identificadores también controlan la visibilidad de los nombres declarados con `typedef`. Por consiguiente, el ejemplo siguiente es válido en C++:

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
```

Otra instancia de ocultación de nombres:

C++

```
// typedefSpecifier1.cpp
typedef char FlagType;

int main()
{
}

void myproc( int )
{
    int FlagType;
}
```

Al declarar un identificador de ámbito local con el mismo nombre que un `typedef`, o al declarar un miembro de una estructura o una unión en el mismo ámbito o en un ámbito interno, se debe indicar el especificador de tipo. Por ejemplo:

C++

```
typedef char FlagType;
const FlagType x;
```

Para reutilizar el nombre `FlagType` para un identificador, un miembro de una estructura o un miembro de una unión, se debe proporcionar el tipo:

C++

```
const int FlagType; // Type specifier required
```

No basta con usar este código:

C++

```
const FlagType; // Incomplete specification
```

El motivo es que se considera que `FlagType` forma parte del tipo, no que sea un identificador que se volverá a declarar. Esta declaración se considera no válida, de forma parecida al código siguiente:

C++

```
int; // Illegal declaration
```

Es posible declarar cualquier tipo con `typedef`, incluidos los tipos de puntero, función y matriz. Se puede declarar un nombre de `typedef` para un puntero a un tipo de estructura o de unión antes de definir el tipo de estructura o de unión, siempre y cuando la definición tenga la misma visibilidad que la declaración.

Ejemplos

Un uso de las declaraciones `typedef` consiste en hacer que las declaraciones sean más uniformes y compactas. Por ejemplo:

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;"
```

Para usar `typedef` con el fin de especificar tipos fundamentales y derivados en la misma declaración, se pueden separar los declaradores mediante comas. Por ejemplo:

C++

```
typedef char CHAR, *PSTR;
```

El ejemplo siguiente proporciona el tipo `DRAWF` para una función que no devuelve ningún valor y que toma dos argumentos `int`:

C++

```
typedef void DRAWF( int, int );
```

Después de la instrucción `typedef` anterior, la declaración

C++

```
DRAWF box;
```

sería equivalente a la declaración

C++

```
void box( int, int );
```

`typedef` se suele combinar con `struct` para declarar y asignar nombres a tipos definidos por el usuario:

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
```

Nueva declaración de definiciones de tipos

La declaración `typedef` se puede usar a fin de volver a declarar el mismo nombre para hacer referencia al mismo tipo. Por ejemplo:

Archivo de origen `file1.h`:

C++

```
// file1.h
typedef char CHAR;
```

Archivo de origen `file2.h`:

C++

```
// file2.h
typedef char CHAR;
```

Archivo de origen `prog.cpp`:

C++

```
// prog.cpp
#include "file1.h"
#include "file2.h" // OK
```

El programa `prog.cpp` incluye dos archivos de encabezado que contienen las declaraciones `typedef` para el nombre `CHAR`. Mientras las declaraciones hagan referencia al mismo tipo, se puede volver a declarar el nombre.

Un `typedef` no puede volver a definir un nombre que se haya declarado como otro tipo. Considere la opción de usar esta alternativa `file2.h`:

C++

```
// file2.h
typedef int CHAR; // Error
```

El compilador genera un error en `prog.cpp` debido al intento de volver a declarar el nombre `CHAR` para hacer referencia a otro tipo. Esta directiva se aplica también a las construcciones como la siguiente:

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
    struct byteregs h;
} REGS;
```

Definiciones de tipos de C++ frente a C

El uso del especificador `typedef` con tipos de clase se admite en gran medida debido a la práctica de ANSI C de declarar las estructuras sin nombre en declaraciones `typedef`. Por ejemplo, muchos programadores de C usan la expresión siguiente:

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;
```

La ventaja de esta declaración es que hace posibles declaraciones como:

C++

```
POINT ptOrigin;
```

en lugar de:

C++

```
struct point_t ptOrigin;
```

En C++, la diferencia entre los nombres de `typedef` y los tipos reales (declarados con las palabras clave `class`, `struct`, `union` y `enum`) es mayor. Aunque la práctica de C de declarar una estructura anónima en una instrucción `typedef` todavía funciona, no proporciona ningún beneficio en cuanto a notación, a diferencia de lo que ocurre en C.

C++

```
// typedef_with_class_types2.cpp
// compile with: /c /W1
typedef struct {
    int POINT();
    unsigned x;
    unsigned y;
} POINT;
```

En el ejemplo anterior, se declara una clase denominada `POINT` mediante la sintaxis `typedef` de la clase sin nombre. `POINT` se trata como un nombre de clase; sin embargo, a los nombres así especificados se aplican las siguientes restricciones:

- El nombre (sinónimo) no se pueden mostrar detrás de un prefijo `class`, `struct` o `union`.
- El nombre no se puede usar como nombre de constructor o destructor dentro de una declaración de clase.

En resumen, esta sintaxis no proporciona ningún mecanismo para la herencia, la construcción o la destrucción.

declaración using

Artículo • 03/03/2023 • Tiempo de lectura: 5 minutos

La declaración `using` introduce un nombre en la región declarativa en la que aparece la declaración "using".

Sintaxis

```
using [typename] nested-name-specifier unqualified-id ;  
using declarator-list ;
```

Parámetros

nested-name-specifier Secuencia de espacios de nombres, clases o nombres de enumeración y operadores de resolución de ámbito (::), terminados mediante un operador de resolución de ámbito. Se puede usar un único operador de resolución de ámbito para introducir un nombre desde el espacio de nombres global. La palabra clave `typename` es opcional y se puede usar para resolver los nombres dependientes cuando se introducen en una plantilla de clase desde una clase base.

unqualified-id Una expresión id-expression no habilitada, que puede ser un identificador, un nombre de operador sobrecargado, un operador literal definido por el usuario o un nombre de función de conversión, un nombre de destructor de clase o un nombre de plantilla y una lista de argumentos.

declarator-list Una lista separada por comas de declaradores `[typename]` *nested-name-specifier* *unqualified-id*, seguido opcionalmente por puntos suspensivos.

Comentarios

Una declaración "using" introduce un nombre no habilitado como sinónimo de una entidad declarada en otro lugar. Permite usar un único nombre obtenido de un espacio de nombres específico sin necesidad de una habilitación explícita en la región de declaración en la que aparece. Esto contrasta con la [directiva "using"](#), que permite utilizar sin habilitación *todos* los nombres de un espacio de nombres. La palabra clave `using` se utiliza también para los [alias de tipo](#).

Ejemplo: declaración using en el campo de clase

Se puede utilizar una declaración using en una definición de clase.

C++

```
// 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()
```

Ejemplo: declaración `using` para declarar un miembro

Cuando se utiliza para declarar un miembro, una declaración `using` debe hacer referencia a un miembro de una clase base.

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()
```

Ejemplo: declaración `using` con habilitación explícita

Se puede hacer referencia a miembros declarados con una declaración `using` mediante una habilitación explícita. El prefijo `::` hace referencia al espacio de nombres global.

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
```

Ejemplo: sinónimos y alias de declaración `using`

Cuando se crea una declaración `using`, el sinónimo creado por la declaración solo hace referencia a las definiciones que son válidas en el lugar de la declaración `using`. Las definiciones que se agregan a un espacio de nombres después de la declaración `using` no son sinónimos válidos.

Un nombre definido por una declaración `using` es un alias para su nombre original. No afecta al tipo, la vinculación u otros atributos de la declaración original.

C++

```
// post_declaration_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);
}
```

Ejemplo: declaraciones locales y declaraciones using

En cuanto a las funciones de espacios de nombres, si se proporciona un conjunto de declaraciones locales y declaraciones using para un único nombre en una región declarativa, todas ellas deben hacer referencia a la misma entidad o todas ellas deben hacer referencia a funciones.

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
}
```

En el ejemplo anterior, la instrucción `using B::i` hace que se declare un segundo `int i` en la función `g()`. La instrucción `using B::f` no entra en conflicto con la función `f(char)` porque los nombres de función introducidos por `B::f` tienen distintos tipos de parámetro.

Ejemplo: declaraciones de función locales y declaraciones `using`

Una declaración de función local no puede tener el mismo nombre y tipo que una función introducida mediante una declaración `using`. Por ejemplo:

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)
}
```

Ejemplo: declaración `using` y herencia

Con respecto a la herencia, cuando una declaración `using` introduce un nombre de una clase base en el ámbito de una clase derivada, las funciones miembro de la clase derivada invalidan las funciones de miembro virtual que tienen los mismos nombres y tipos de argumento en la clase base.

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)

```

Ejemplo: accesibilidad a la declaración **using**

Todas las instancias de un nombre mencionado en una declaración using deben ser accesibles. En concreto, si una clase derivada utiliza una declaración using para tener acceso a un miembro de una clase base, el nombre de miembro debe ser accesible. Si el nombre es el de una función miembro sobrecargada, todas las funciones enumeradas deben ser accesibles.

Consulte [Control de acceso a miembros](#) para más información sobre la accesibilidad de los miembros.

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
};
```

Consulte también

[Espacios de nombres](#)

[Palabras clave](#)

volatile (C++)

Artículo • 26/09/2022 • Tiempo de lectura: 3 minutos

Calificador de tipo que puede utilizar para declarar que el hardware puede modificar un objeto en el programa.

Sintaxis

```
volatile declarator ;
```

Comentarios

Puede usar el modificador [/volatile](#) del compilador para modificar cómo interpreta el compilador esta palabra clave.

Visual Studio interpreta la palabra clave `volatile` de manera diferente según la arquitectura de destino. En el caso de ARM, si no se especifica ninguna opción [/volatile](#) del compilador, este funciona como si se hubiera especificado [/volatile:iso](#). En las arquitecturas distintas de ARM, si no se especifica ninguna opción [/volatile](#) del compilador, el compilador funciona como si se hubiera especificado [/volatile:ms](#). Por tanto, para otras arquitecturas que no sean ARM, se recomienda encarecidamente especificar [/volatile:iso](#) y usar tipos primitivos de sincronización e intrínsecos del compilador explícitos al tratar con memoria que se comparte entre distintos subprocessos.

Se puede usar el calificador `volatile` para proporcionar acceso a ubicaciones de memoria empleadas por procesos asincrónicos como controladores de interrupciones.

Cuando se usa `volatile` en una variable que tiene también la palabra clave [__restrict](#), tiene prioridad `volatile`.

Si un miembro `struct` está marcado como `volatile`, `volatile` se propaga a toda la estructura. Si una estructura no tiene una longitud que se pueda copiar en la arquitectura actual mediante una instrucción, se puede perder completamente `volatile` en esa estructura.

La palabra clave `volatile` puede no tener ningún efecto sobre un campo si se cumple una de las condiciones siguientes:

- La longitud del campo volátil supera el tamaño máximo que se puede copiar en la arquitectura actual mediante una instrucción.
- La longitud del `struct` contenedor exterior (o si es miembro de un `struct` posiblemente anidado) supera el tamaño máximo que se puede copiar en la arquitectura actual mediante una instrucción.

Aunque el procesador no reordena los accesos a memoria que no pueden almacenarse en la memoria caché, las variables que no pueden almacenarse en la memoria caché deben marcarse como `volatile` para garantizar que el compilador no reordene los accesos a memoria.

Los objetos declarados como `volatile` no se usan en ciertas optimizaciones porque sus valores pueden cambiar en cualquier momento. El sistema lee siempre el valor actual de un objeto volátil cuando se solicita, incluso aunque una instrucción anterior pidiera un valor del mismo objeto. Además, el valor del objeto se escribe inmediatamente en la asignación.

Conformidad con ISO

Si está familiarizado con la palabra clave `volatile` de C# o con el comportamiento de `volatile` en versiones anteriores del compilador de Microsoft C++ (MSVC), tenga en cuenta que la palabra clave `volatile` del estándar C++11 de ISO es diferente y se admite en MSVC cuando se especifica la opción `/volatile:iso` del compilador. (Para ARM, se especifica de forma predeterminada). La palabra clave `volatile` en código del estándar C++11 de ISO se usa únicamente para el acceso de hardware; no la use para la comunicación entre subprocessos. Para la comunicación entre subprocessos, emplee mecanismos como `std::atomic<T>` de la [biblioteca estándar de C++](#).

Fin de la conformidad con ISO

Específicos de Microsoft

Cuando se usa la opción `/volatile:ms` del compilador (de forma predeterminada, cuando el destino es una arquitectura distinta de ARM), el compilador genera código adicional para mantener el orden entre las referencias a objetos volátiles y el orden de las referencias a otros objetos globales. En concreto:

- Una escritura en un objeto volátil (también conocida como escritura volátil) tiene liberación de semántica; es decir, una referencia a un objeto global o estático que

se produce antes que una escritura en un objeto volátil en la secuencia de instrucciones se realice antes que esa escritura volátil en el binario compilado.

- Una lectura de un objeto volátil (también conocida como lectura volátil) tiene adquisición de semántica; es decir, una referencia a un objeto global o estático que se produce después que una lectura de memoria volátil en la secuencia de instrucciones se realice después de esa lectura volátil en el binario compilado.

Esto permite utilizar objetos volátiles para bloqueos y liberaciones de memoria en aplicaciones multiproceso.

ⓘ Nota

Cuando se basa en la seguridad mejorada que se proporciona cuando se usa la opción `/volatile:ms` del compilador, el código es no portable.

FIN de Específicos de Microsoft

Consulte también

[Palabras clave](#)

[const](#)

[Punteros const y volatile](#)

decltype (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 6 minutos

El especificador de tipo `decltype` produce el tipo de una expresión especificada. El especificador de tipo `decltype`, junto con la [palabra clave auto](#), es útil sobre todo para los desarrolladores que crean bibliotecas de plantilla. Use `auto` y `decltype` para declarar una plantilla de función cuyo tipo de valor devuelto dependa de los tipos de sus argumentos de plantilla. O bien, use `auto` y `decltype` para declarar una plantilla de función que encapsula una llamada a otra función y, a continuación, devuelva el tipo de valor devuelto de la función ajustada.

Sintaxis

```
decltype( expression )
```

Parámetros

expression

Expresión. Para más información, consulte [Expresiones](#).

Valor devuelto

Tipo del parámetro *expression*.

Comentarios

El especificador de tipo `decltype` se admite en Visual Studio 2010 o versiones posteriores, y se puede usar con código nativo o administrado. `decltype(auto)` (C++14) se admite en Visual Studio de 2015 y versiones posteriores.

El compilador usa las siguientes reglas para determinar el tipo del *expression* parámetro .

- Si el *expression* parámetro es un identificador o un [acceso de miembro de clase](#), `decltype(expression)` es el tipo de la entidad denominada por `.expression`. Si no hay ninguna entidad de este tipo o el *expression* parámetro asigna un nombre a un conjunto de funciones sobrecargadas, el compilador genera un mensaje de error.

- Si el `expression` parámetro es una llamada a una función o a una función de operador sobrecargada, `decltype(expression)` es el tipo de valor devuelto de la función. Los paréntesis alrededor de un operador sobrecargado se omiten.
- Si el `expression` parámetro es un **valor rvalue**, `decltype(expression)` es el tipo de `expression`. Si el `expression` parámetro es un **valor lvalue**, `decltype(expression)` es una **referencia lvalue** al tipo de `expression`.

En el ejemplo de código siguiente se muestran algunos usos del especificador de tipo `decltype`. En primer lugar, supongamos que ha codificado las siguientes instrucciones.

C++

```
int var;
const int&& fx();
struct A { double x; }
const A* a = new A();
```

A continuación, examine los tipos devueltos por las cuatro instrucciones `decltype` de la tabla siguiente.

| . | Tipo | Notas |
|-----------------------------------|----------------------------------|--|
| <code>decltype(fx());</code> | <code>const int&&</code> | Una referencia rvalue a <code>const int</code> . |
| <code>decltype(var);</code> | <code>int</code> | El tipo de variable <code>var</code> . |
| <code>decltype(a->x);</code> | <code>double</code> | El tipo del acceso a miembros. |
| <code>decltype((a->x));</code> | <code>const double&</code> | Los paréntesis internos hacen que la instrucción se evalúe como una expresión en lugar de como un acceso a miembros. Y como <code>a</code> se declara como un puntero <code>const</code> , el tipo es una referencia a <code>const double</code> . |

decltype y auto

En C++14, puede usar `decltype(auto)` sin tipo de valor devuelto final para declarar una plantilla de función cuyo tipo de valor devuelto depende de los tipos de sus argumentos de plantilla.

En C++11, puede usar el `decltype` especificador de tipo en un tipo de valor devuelto final, junto con la `auto` palabra clave, para declarar una plantilla de función cuyo tipo de valor devuelto depende de los tipos de sus argumentos de plantilla. Por ejemplo,

considere el siguiente ejemplo de código en el que el tipo de valor devuelto de la plantilla de función depende de los tipos de los argumentos de plantilla. En el ejemplo de código, el `UNKNOWN` marcador de posición indica que no se puede especificar el tipo de valor devuelto.

C++

```
template<typename T, typename U>
UNKNOWN func(T&& t, U&& u){ return t + u; };
```

La introducción del `decltype` especificador de tipo permite a un desarrollador obtener el tipo de la expresión que devuelve la plantilla de función. Utilice la *sintaxis de declaración de función alternativa* que se muestra más adelante, la palabra clave `auto` y el especificador de tipo `decltype` para declarar un tipo de valor devuelto *especificado en tiempo de compilación*. El tipo de valor devuelto especificado en tiempo de espera se determina cuando se compila la declaración, en lugar de cuando se codifica.

El prototipo siguiente muestra la sintaxis de una declaración de función alternativa. Los `const` calificadores y `volatile` y la `throw` especificación de excepción son opcionales . El `function_body` marcador de posición representa una instrucción compuesta que especifica lo que hace la función. Como procedimiento recomendado de codificación, el `expression` marcador de posición de la `decltype` instrucción debe coincidir con la expresión especificada por la `return` instrucción, si existe, en `.function_body`

```
auto function_name ( parameters ) const volatile Optar -> decltype( expression ) noexcept Optar { function_body };
```

En el ejemplo de código siguiente, el tipo de valor devuelto especificado en tiempo de ejecución de la `myFunc` plantilla de función viene determinado por los tipos de los `t` argumentos de plantilla y `u` . Como procedimiento de programación recomendado, en el ejemplo de código también se utilizan referencias rvalue y la plantilla de función `forward`, que admiten el *reenvío directo*. Para más información, vea [Declarador de referencia de rvalue&&](#).

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 y funciones de reenvío (C++11)

Las funciones de reenvío encapsulan llamadas a otras funciones. Considere una plantilla de función que reenvía sus argumentos, o los resultados de una expresión en la que participan estos argumentos, a otra función. Además, la función de reenvío devuelve el resultado de la llamada a otra función. En este escenario, el tipo de valor devuelto de la función de reenvío debe ser el mismo que el tipo de valor devuelto de la función encapsulada.

En este escenario, no se puede escribir una expresión de tipo adecuada sin el especificador de `decltype` tipos. El `decltype` especificador de tipos habilita las funciones de reenvío genérico porque no pierde la información necesaria sobre si una función devuelve un tipo de referencia. Para obtener un ejemplo de código de una función de reenvío, consulte el ejemplo de plantilla de función anterior `myFunc`.

Ejemplos

En el ejemplo de código siguiente se declara el tipo de valor devuelto especificado en tiempo de ejecución de la plantilla `Plus()` de función. La función `Plus` procesa sus operandos con la sobrecarga `operator+`. Por lo tanto, la interpretación del operador más (`+`) y el tipo de valor devuelto de la `Plus` función depende de los tipos de los argumentos de función.

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 y versiones posteriores: el compilador analiza los argumentos `decltype` cuando se declaran las plantillas en lugar de crear instancias. Por lo tanto, si se encuentra una especialización no dependiente en el `decltype` argumento , no se aplazará al tiempo de creación de instancias; se procesará inmediatamente y se diagnosticarán los errores resultantes en ese momento.

En el ejemplo siguiente se muestra este tipo de error del compilador que se genera en el punto de declaración:

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");
```

Requisitos

Visual Studio 2010 o versiones posteriores.

`decltype(auto)` requiere Visual Studio 2015 o una versión posterior.

Atributos de C++

Artículo • 03/03/2023 • Tiempo de lectura: 7 minutos

C++ estándar define un conjunto común de atributos. También permite a los proveedores de compiladores definir sus propios atributos dentro de un espacio de nombres específico del proveedor. Sin embargo, los compiladores solo son necesarios para reconocer los atributos definidos en el estándar.

En algunos casos, los atributos estándar se superponen con parámetros `_declspec` específicos del compilador. En Microsoft C++, puede usar el atributo `[[deprecated]]` en lugar de usar `_declspec(deprecated)`. Cualquier compilador compatible reconoce el atributo `[[deprecated]]`. Para todos los demás parámetros `_declspec`, como `dllimport` y `dllexport`, hasta ahora no hay ningún atributo equivalente, por lo que debe seguir usando la sintaxis `_declspec`. Los atributos no afectan al sistema de tipos y no cambian el significado de un programa. Los compiladores omiten los valores de atributo que no reconocen.

Visual Studio 2017, versión 15.3 y posteriores (disponible con `/std:c++17` y versiones posteriores): en el ámbito de una lista de atributos, puede especificar el espacio de nombres de todos los nombres con un único introductor `using`:

C++

```
void g() {
    [[using rpr: kernel, target(cpu,gpu)]] // equivalent to [[ rpr::kernel,
    rpr::target(cpu,gpu) ]]
    do task();
}
```

Atributos estándar de C++

En C++11, los atributos proporcionan una manera estandarizada de anotar construcciones de C++ (incluidas, entre otras, clases, funciones, variables y bloques) con información adicional. Los atributos pueden o no ser específicos del proveedor. Un compilador puede usar esta información para generar mensajes informativos o para aplicar lógica especial al compilar el código con atributos. El compilador omite los atributos que no reconoce, lo que significa que no puede definir sus propios atributos personalizados mediante esta sintaxis. Los atributos se incluyen entre corchetes dobles:

C++

```
[[deprecated]]  
void Foo(int);
```

Los atributos representan una alternativa estandarizada a extensiones específicas del proveedor, como directivas `#pragma`, `__declspec()` (Visual C++) o `__attribute__` (GNU). Sin embargo, para la mayoría de los propósitos, tendrá que usar las construcciones específicas del proveedor. Actualmente, el estándar especifica los siguientes atributos que debe reconocer un compilador conforme.

[[noreturn]]

El `[[noreturn]]` atributo especifica que una función nunca devuelve; en otras palabras, siempre produce una excepción o se cierra. El compilador puede ajustar sus reglas de compilación para las entidades `[[noreturn]]`.

[[carries_dependency]]

El `[[carries_dependency]]` atributo especifica que la función propaga el orden de dependencia de datos para la sincronización de subprocessos. El atributo se puede aplicar a uno o varios parámetros para especificar que el argumento pasado lleva una dependencia en el cuerpo de la función. El atributo se puede aplicar a la propia función para especificar que el valor devuelto lleva una dependencia fuera de la función. El compilador puede usar esta información para generar código más eficaz.

[[deprecated]]

Visual Studio 2015 y versiones posteriores: El `[[deprecated]]` atributo especifica que una función no está pensada para su uso. O bien, es posible que no exista en versiones futuras de una interfaz de biblioteca. El `[[deprecated]]` atributo se puede aplicar a la declaración de una clase, un `typedef-name`, una variable, un miembro de datos no estático, una función, un espacio de nombres, una enumeración, un enumerador o una especialización de plantilla. El compilador puede usar este atributo para generar un mensaje informativo cuando el código de cliente intenta llamar a la función. Cuando el compilador de C++ Microsoft detecta el uso de un `[[deprecated]]` elemento, genera la advertencia del compilador [C4996](#).

[[fallthrough]]

Visual Studio 2017 y versiones posteriores: (disponible con [/std:c++17](#) y versiones posteriores). El atributo `[[fallthrough]]` se puede usar en el contexto de las instrucciones `switch` como una sugerencia para el compilador (o cualquier persona que lea el código) de que el comportamiento de fallthrough está previsto. Actualmente, el compilador de Microsoft C++ no advierte sobre el comportamiento de paso explícito, por lo que este atributo no tiene ningún efecto sobre el comportamiento del compilador.

[[nodiscard]]

Visual Studio 2017, versión 15.3 y posteriores: (disponible con [/std:c++17](#) y versiones posteriores). Especifica que el valor devuelto de una función no está destinado a ser descartado. Genera la advertencia [C4834](#), como se muestra en este ejemplo:

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;
}
```

[[maybe_unused]]

Visual Studio 2017, versión 15.3 y posteriores: (disponible con [/std:c++17](#) y versiones posteriores). El `[[maybe_unused]]` atributo especifica que una variable, función, clase, typedef, miembro de datos no estático, enumeración o especialización de plantilla puede no usarse intencionadamente. El compilador no advierte cuando no se usa una entidad marcada como `[[maybe_unused]]`. Una entidad declarada sin el atributo se puede volver a declarar posteriormente con el atributo y viceversa. Una entidad se considera *marcada* después de que se analiza su primera declaración marcada como `[[maybe_unused]]` y para el resto de la unidad de traducción actual.

[[likely]]

Visual Studio 2019, versión 16.6 y posteriores: (disponible con [/std:c++20](#) y versiones posteriores). El `[[likely]]` atributo especifica una sugerencia al compilador de que la ruta de acceso del código para la etiqueta o instrucción con atributos es más probable

que se ejecute que las alternativas. En el compilador de Microsoft, el atributo `[[likely]]` marca los bloques como "código frecuente", lo que incrementa una puntuación de optimización interna. La puntuación se incrementa más al optimizar la velocidad y no tanto al optimizar el tamaño. La puntuación neta afecta a la probabilidad de insertar, desenrollar bucles y vectorizar optimizaciones. El efecto de `[[likely]]` y `[[unlikely]]` es similar a la característica [Optimización guiada por perfiles](#), pero limitado en el ámbito a la unidad de traducción actual. La optimización del reordenamiento de bloques aún no se ha implementado para este atributo.

`[[unlikely]]`

Visual Studio 2019, versión 16.6 y posteriores: (disponible con `/std:c++20` y versiones posteriores). El `[[unlikely]]` atributo especifica una sugerencia al compilador de que es menos probable que se ejecute la ruta de acceso del código de la etiqueta o instrucción con atributos que las alternativas. En el compilador de Microsoft, el atributo `[[unlikely]]` marca los bloques como "código poco frecuente", lo que reduce una puntuación de optimización interna. La puntuación se reduce más al optimizar el tamaño y no tanto al optimizar la velocidad. La puntuación neta afecta a la probabilidad de insertar, desenrollar bucles y vectorizar optimizaciones. La optimización del reordenamiento de bloques aún no se ha implementado para este atributo.

Atributos específicos de Microsoft

`[[gsl::suppress(rules)]]`

El atributo específico `[[gsl::suppress(rules)]]` del Microsoft se usa para suprimir las advertencias de los comprobadores que aplican reglas de la [Biblioteca de compatibilidad de directrices \(GSL\)](#) en el código. Por ejemplo, considere este fragmento de código:

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
    }
}
```

En el ejemplo se generan estas advertencias:

- [C26494](#) (Regla de tipo 5: Inicializar siempre un objeto).
- [C26485](#) (Regla de límites 3: sin matriz a decremento de puntero).
- [C26481](#) (Regla de límites 1: No usar aritmética de punteros. Use span en su lugar).

Las dos primeras advertencias se activan al compilar este código con la herramienta de análisis de código CppCoreCheck instalada y activada. Pero la tercera advertencia no se desencadena debido al atributo. Puede suprimir todo el perfil de límites escribiendo `[[gsl::suppress(bounds)]]` sin incluir un número de regla específico. Las directrices de C++ Core Guidelines están diseñadas para ayudarle a escribir código mejor y más seguro. El atributo `suppress` facilita la desactivación de las advertencias cuando no se desean.

[[msvc::intrinsic]]

El atributo específico `[[msvc::intrinsic]]` del Microsoft indica al compilador que inserte una metafunción que actúa como una conversión con nombre del tipo de parámetro al tipo de valor devuelto. Cuando el atributo está presente en una definición de función, el compilador reemplaza todas las llamadas a esa función por una conversión simple. El `[[msvc::intrinsic]]` atributo está disponible en visual Studio 2022 versión 17.5 preview 2 y versiones posteriores. Este atributo solo se aplica a la función específica que la sigue.

El `[[msvc::intrinsic]]` atributo tiene dos restricciones en la función a la que se aplica:

1. La función no puede ser recursiva; su cuerpo solo debe tener una instrucción `return` con una conversión.
2. La función solo puede aceptar un único parámetro.
3. La opción del compilador `/permissive-` no es necesaria. (Las `/std:c++20` opciones y posteriores implican `/permissive-` de forma predeterminada).

Ejemplo

En este código de ejemplo, el `[[msvc::intrinsic]]` atributo aplicado a la `my_move` función hace que el compilador reemplace las llamadas a la función por la conversión estática insertada en su cuerpo:

```
template <typename T>
[[msvc::intrinsics]] T&& my_move(T&& t) { return static_cast<T&&>(t); }

void f() {
    int i = 0;
    i = my_move(i);
}
```

[[msvc::no_tls_guard]]

El atributo específico `[[msvc::no_tls_guard]]` del Microsoft deshabilita las comprobaciones de inicialización en el primer acceso a variables locales de subprocessos en archivos DLL. Las comprobaciones están habilitadas de forma predeterminada en el código compilado con Visual Studio 2019 versión 16.5 y versiones posteriores. Este atributo solo se aplica a la variable específica que la sigue. Para deshabilitar las comprobaciones globalmente, use la [/Zc:tlsGuards-](#) opción del compilador .

Operadores integrados de C++, precedencia y asociatividad

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

El lenguaje C++ incluye todos los operadores de C y agrega varios operadores nuevos. Los operadores especifican una evaluación que se realizará en uno o más operandos.

Precedencia y asociatividad

La *prioridad* de los operadores especifica el orden en que se realizan las operaciones en las expresiones que contienen más de un operador. La *asociatividad* de los operadores especifica si, en una expresión que contiene varios operadores con la misma prioridad, un operando se agrupa con el de su izquierda o con el de su derecha.

Ortografías alternativas

C++ especifica ortografías alternativas para algunos operadores. En C, las ortografías alternativas se proporcionan como macros en el encabezado de `<iso646.h>`. En C++, estas alternativas son palabras clave y el uso de `<iso646.h>` o el equivalente en C++ `<ciso646>` está en desuso. En Microsoft C++, se requiere la opción del compilador `/permissive-` o `/Za` para poder habilitar las ortografías alternativas.

Tabla de prioridad y asociatividad de los operadores de C++

La tabla siguiente muestra la prioridad y la asociatividad de los operadores de C++ (de mayor a menor prioridad). Los operadores que tienen el mismo número de prioridad tienen la misma prioridad, a menos que se fuerce otra relación explícitamente mediante paréntesis.

| Descripción del operador | Operador | Alternativa |
|--|------------------------|-------------|
| Precedencia de grupo 1, sin asociatividad | | |
| Resolución de ámbito | <code>::</code> | |
| Precedencia de grupo 2, asociatividad de izquierda a derecha | | |
| Selección de miembro (objeto o puntero) | <code>. o -></code> | |

| Descripción del operador | Operador | Alternativa |
|---|------------------|--------------------|
| Subíndice de matriz | [] | |
| Llamada a función | () | |
| Incremento de postfijo | ++ | |
| Decremento de postfijo | -- | |
| Nombre de tipo | typeid | |
| Conversión de tipos constante | const_cast | |
| Conversión de tipos dinámica | dynamic_cast | |
| Conversión de tipos reinterpretada | reinterpret_cast | |
| Conversión de tipos estática | static_cast | |
| Precedencia de grupo 3, asociatividad de derecha a izquierda | | |
| Tamaño de objeto o tipo | sizeof | |
| Incremento de prefijo | ++ | |
| Decremento de prefijo | -- | |
| Complemento a uno | ~ | compl |
| NOT lógico | ! | not |
| Negación unaria | - | |
| Suma unaria | + | |
| Dirección de | & | |
| Direccionamiento indirecto | * | |
| Crear objeto | new | |
| Destruir objeto | delete | |
| Conversión | () | |
| Precedencia de grupo 4, asociatividad de izquierda a derecha | | |
| Puntero a miembro (objetos o punteros) | .* o ->* | |
| Precedencia de grupo 5, asociatividad de izquierda a derecha | | |
| Multiplicación | * | |

| Descripción del operador | Operador | Alternativa |
|---|-----------------|--------------------|
| División | / | |
| Módulo | % | |
| Precedencia de grupo 6, asociatividad de izquierda a derecha | | |
| Suma | + | |
| Resta | - | |
| Precedencia de grupo 7, asociatividad de izquierda a derecha | | |
| Desplazamiento a la izquierda | << | |
| Desplazamiento a la derecha | >> | |
| Precedencia de grupo 8, asociatividad de izquierda a derecha | | |
| Menor que | < | |
| Mayor que | > | |
| Menor o igual que | <= | |
| Mayor o igual que | >= | |
| Precedencia de grupo 9, asociatividad de izquierda a derecha | | |
| Igualdad | == | |
| Desigualdad | != | not_eq |
| Precedencia de grupo 10 asociatividad de izquierda a derecha | | |
| AND bit a bit | & | bitand |
| Precedencia de grupo 11, asociatividad de izquierda a derecha | | |
| OR exclusivo bit a bit | ^ | xor |
| Precedencia de grupo 12, asociatividad de izquierda a derecha | | |
| OR inclusivo bit a bit | | bitor |
| Precedencia de grupo 13, asociatividad de izquierda a derecha | | |
| Y lógico | && | and |
| Precedencia de grupo 14, asociatividad de izquierda a derecha | | |
| O lógico | | or |

| Descripción del operador | Operador | Alternativa |
|---|-----------------|--------------------|
| Precedencia de grupo 15, asociatividad de derecha a izquierda | | |
| Condicional | ? : | |
| Cesión | = | |
| Asignación y multiplicación | *= | |
| Asignación y división | /= | |
| Asignación y módulo | %= | |
| Asignación y suma | + = | |
| Asignación y resta | - = | |
| Asignación y desplazamiento a la izquierda | <<= | |
| Asignación y desplazamiento a la derecha | >>= | |
| Asignación AND bit a bit | &= | and_eq |
| Asignación y OR inclusivo bit a bit | = | or_eq |
| Asignación OR exclusivo bit a bit | ^ = | xor_eq |
| Expresión Throw | throw | |
| Precedencia de grupo 16, asociatividad de izquierda a derecha | | |
| Coma | , | |

Consulte también

[Sobrecarga de operadores](#)

Operador

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

El operador `devuelve la alineación en bytes del tipo especificado como un valor de tipo .`

Sintaxis

C++

Comentarios

Por ejemplo:

| Expression | Valor |
|---|-------|
| <code)<="" alignof(="" char="" code=""></code> | 1 |
| <code)<="" alignof(="" code="" short=""></code> | 2 |
| <code)<="" alignof(="" code="" int=""></code> | 4 |
| <code)<="" alignof(="" code="" long=""></code> | 8 |
| <code)<="" alignof(="" code="" float=""></code> | 4 |
| <code)<="" alignof(="" code="" double=""></code> | 8 |

El valor `es igual que el valor de para los tipos básicos. Consideré, no obstante, este ejemplo:`

C++

```
typedef struct { int a; double b; } S;
// alignof(S) == 8
```

En este caso, el valor `es el requisito de alineación del elemento más grande de la estructura.`

De igual forma, para

C++

```
typedef __declspec(align(32)) struct { int a; } S;
```

`alignof(S)` es igual a `32`.

Un uso para `alignof` sería como parámetro para una de sus propias rutinas de asignación de memoria. Por ejemplo, dada la siguiente estructura definida `S`, podría llamar a una rutina de asignación de memoria denominada `aligned_malloc` para asignar memoria en un límite de alineación determinado.

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));
```

Para obtener más información sobre la modificación de la alineación, vea:

- [pack](#)
- [align](#)
- [_unaligned](#)
- [/Zp \(Alineación de miembros de struct\)](#)
- [Ejemplos de alineación de estructuras x64](#)

Para obtener más información sobre las diferencias de la alineación en código para x86 y x64, vea:

- [Conflictos con el compilador de x86](#)

Específico de Microsoft

`alignof` y `_alignof` son sinónimos en el compilador de Microsoft. Antes de que se convirtiera en parte del estándar en C++11, el operador `_alignof` específico de Microsoft ofrecía esta funcionalidad. Para la máxima portabilidad, use el operador `alignof` en lugar del operador `_alignof` específico de Microsoft.

A efectos de compatibilidad con versiones anteriores, `_alignof` es un sinónimo de `_alignof` a menos que se especifique la opción del compilador [/Za \(Deshabilitar extensiones de lenguaje\)](#).

Vea también

Expresiones con operadores unarios

Palabras clave

Operador `_uuidof`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Recupera el GUID asociado a la expresión.

Sintaxis

```
_uuidof ( expression )
```

Comentarios

La *expresión* puede ser un nombre de tipo, un puntero, una referencia o una matriz de ese tipo, una plantilla especializada en estos tipos, o una variable de estos tipos. El argumento es válido siempre y cuando el compilador pueda utilizarlo para encontrar el GUID asociado.

Un caso especial de este intrínseco es cuando se proporciona **0** o **NULL** como argumento. En este caso, `_uuidof` devolverá un GUID compuesto de ceros.

Utilice esta palabra clave para extraer el GUID asociado a lo siguiente:

- Un objeto por el atributo extendido [uuid](#).
- Un bloque de biblioteca creado con el atributo [module](#).

ⓘ Nota

En una compilación de depuración, `_uuidof` siempre inicializa un objeto dinámicamente (en tiempo de ejecución). En una compilación de versión, `_uuidof` puede inicializar estáticamente (en tiempo de compilación) un objeto.

A efectos de compatibilidad con versiones anteriores, `_uuidof` es un sinónimo de `_uuidof` a menos que se especifique la opción del compilador [/Za \(Deshabilitar extensiones de lenguaje\)](#).

Ejemplo

El código siguiente (compilado con ole32.lib) mostrará el uuid de un bloque de biblioteca creado con el atributo module:

```
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);  
}
```

Comentarios

En aquellos casos en los que el nombre de la biblioteca ya no se encuentre en el ámbito, puede utilizar `__LIBID_` en lugar de `__uuidof`. Por ejemplo:

```
C++  
  
StringFromCLSID(__LIBID_, &lpolestr);
```

FIN de Específicos de Microsoft

Vea también

[Expresiones con operadores unarios](#)

[Palabras clave](#)

Operadores de adición: + y -

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Sintaxis

```
expression + expression  
expression - expression
```

Comentarios

Los operadores aditivos son:

- Adición (+)
- Resta (-)

Estos operadores binarios tienen asociatividad de izquierda a derecha.

Los operadores aditivos toman operandos de tipos aritméticos o de puntero. El resultado del operador de suma (+) es la suma de los operandos. El resultado del operador de resta (-) es la diferencia entre los operandos. Si uno o ambos operandos son punteros, deben ser punteros a objetos, no a funciones. Si ambos operandos son punteros, los resultados no son significativos a menos que ambos sean punteros a objetos de la misma matriz.

Los operadores aditivos toman operandos de tipos *aritméticos*, *enteros* y *escalares*. Se definen en la tabla siguiente.

Tipos utilizados con operadores de suma

| Tipo | Significado |
|--------------------|--|
| <i>aritméticos</i> | Los tipos enteros y de punto flotante se denominan colectivamente tipos "aritméticos". |
| <i>enteros</i> | Los tipos char e int de todos los tamaños (long, short) y las enumeraciones son tipos "enteros". |
| <i>escalares</i> | Los operandos escalares son operandos de tipo aritmético o puntero. |

Las combinaciones válidas para estos operadores son:

aritméticos + aritméticos

escalares + enteros

enteros + escalares

aritméticos - aritméticos

escalares - escalares

Tenga en cuenta que la suma y resta no son operaciones equivalentes.

Si ambos operandos son de tipo aritmético, las conversiones descritas en [Conversiones estándar](#) se aplican a los operandos y el resultado es de tipo convertido.

Ejemplo

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;
}
```

Adición de puntero

Si uno de los operandos de una operación de suma es un puntero a una matriz de objetos, el otro debe ser de tipo entero. El resultado es un puntero del mismo tipo que el puntero original y que apunta a otro elemento de la matriz. En el siguiente fragmento de código se muestra este concepto:

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;
}
```

Aunque el valor entero 1 se suma a `pIntArray`, eso no significa "sumar 1 a la dirección", sino más bien "ajustar el puntero para que apunte al siguiente objeto de la matriz", que resulta estar alejado 2 bytes (o `sizeof(int)`).

ⓘ Nota

El código con la forma `pIntArray = pIntArray + 1` raramente aparece en programas de C++; para realizar un incremento, son preferibles estas formas:
`pIntArray++` o `pIntArray += 1`.

Resta de puntero

Si ambos operandos son punteros, el resultado de la resta es la diferencia (en elementos de matriz) entre los operandos. La expresión de resta produce un resultado entero con signo de tipo `ptrdiff_t` (definido en el archivo de inclusión estándar `<stddef.h>`).

Uno de los operandos puede ser de tipo entero, siempre y cuando sea el segundo operando. El resultado de la resta es del mismo tipo que el puntero original. El valor de la resta es un puntero al elemento de la matriz ($n - i$), donde n es el elemento al que señala el puntero original e i es el valor entero del segundo operando.

Consulte también

[Expresiones con operadores binarios](#)

[Operadores integrados de C++, precedencia y asociatividad](#)

[Operadores de adición de C](#)

Operador Address-of: &

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Sintaxis

`address-of-expression:`

`& cast-expression`

Comentarios

El operador unario address-of (`&`) devuelve la dirección de (es decir, un puntero a) su operando. El operando del operador address-of puede ser un designador de función de un lvalue que se refiere a un objeto que no es un campo de bit.

El operador address-of solo se puede aplicar a ciertas expresiones lvalue: ya sea variables de tipo fundamental, de estructura, de clase o de unión, o a referencias de matriz de subíndice. En estas expresiones, se puede agregar o quitar de la expresión address-of una expresión constante (una que no incluya el operador address-of).

Cuando se aplica a funciones o lvalues, el resultado de la expresión es un tipo de puntero (un rvalue) derivado del tipo del operando. Por ejemplo, si el operando es de tipo `char`, el resultado de la expresión es de tipo puntero a `char`. El operador address-of, aplicado a objetos `const` o `volatile`, se evalúa como `const type *` o `volatile type *`, donde `type` es el tipo del objeto original.

La dirección de una función sobrecargada se puede tomar solo cuando está claro a qué versión de la función se hace referencia. Para más información sobre cómo obtener la dirección de una función sobrecargada concreta, consulte [Sobrecarga de funciones](#).

Cuando el operador address-of se aplica a un nombre calificado, el resultado depende de si el *qualified-name* especifica un miembro estático. En ese caso, el resultado es un puntero al tipo especificado en la declaración del miembro. Si un miembro no es estático, el resultado es un puntero al miembro *name* de la clase que se indica mediante *qualified-class-name*. Para obtener más información sobre *qualified-class-name*, consulte [Expresiones principales](#).

Ejemplo: dirección del miembro estático

En el fragmento de código siguiente se muestra cómo difiere el resultado del operador address-of en función de si el miembro de clase es estático:

```
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  
}
```

En este ejemplo, la expresión `&PTM::fValue` proporciona el tipo `float *` en lugar de `float PTM::*`, porque `fValue` es un miembro estático.

Ejemplo: dirección de un tipo de referencia

Al aplicar el operador address-of a un tipo de referencia se obtiene el mismo resultado que al aplicar el operador al objeto al que se enlaza la referencia. Por ejemplo:

```
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
```

Ejemplo: dirección de función como parámetro

En el ejemplo siguiente se usa el operador address-of para pasar un argumento de puntero a una función:

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

Consulte también

[Expresiones con operadores unarios](#)

[Operadores integrados de C++, precedencia y asociatividad](#)

[Declarador de referencia a lvalue: &](#)

[Operadores de direccionamiento indirecto y address-of](#)

Operadores de asignación

Artículo • 03/03/2023 • Tiempo de lectura: 6 minutos

Sintaxis

`expression assignment-operator expression`

`assignment-operator`: uno de

= *= /= %= += -= <<= >>= &= ^= |=

Comentarios

Los operadores de asignación almacenan un valor en el objeto especificado por el operando izquierdo. Hay dos tipos de operaciones de asignación:

- *asignación simple*, en la que el valor del segundo operando se almacena en el objeto especificado por el primer operando.
- *asignación compuesta*, en la que se realiza una operación aritmética, de desplazamiento o bit a bit antes de almacenar el resultado.

Todos los operadores de asignación de la tabla siguiente, salvo el operador =, son operadores de asignación compuesta.

Tabla de operadores de asignación

| Operator | Significado |
|----------|--|
| = | Almacena el valor del segundo operando en el objeto especificado por el primer operando (asignación simple). |
| *= | Multiplica el valor del primer operando por el valor del segundo operando; almacena el resultado en el objeto especificado por el primer operando. |
| /= | Divide el valor del primer operando por el valor del segundo operando; almacena el resultado en el objeto especificado por el primer operando. |
| %= | Toma el módulo del primer operando especificado por el valor del segundo operando; almacena el resultado en el objeto especificado por el primer operando. |
| += | Suma el valor del segundo operando al valor del primer operando; almacena el resultado en el objeto especificado por el primer operando. |

| Operator | Significado |
|------------------------|--|
| <code>-=</code> | Resta el valor del segundo operando del valor del primer operando; almacena el resultado en el objeto especificado por el primer operando. |
| <code><<=</code> | Desplaza a la izquierda el valor del primer operando el número de bits especificado por el valor del segundo operando; almacena el resultado en el objeto especificado por el primer operando. |
| <code>>>=</code> | Desplaza a la derecha el valor del primer operando el número de bits especificado por el valor del segundo operando; almacena el resultado en el objeto especificado por el primer operando. |
| <code>&=</code> | Obtiene el AND bit a bit del primer y el segundo operandos; almacena el resultado en el objeto especificado por el primer operando. |
| <code>^=</code> | Obtiene el OR exclusivo bit a bit del primer y el segundo operandos; almacena el resultado en el objeto especificado por el primer operando. |
| <code> =</code> | Obtiene el OR inclusivo bit a bit del primer y el segundo operandos; almacena el resultado en el objeto especificado por el primer operando. |

Palabras clave de operador

Tres de los operadores de asignación compuesta tienen equivalentes de palabras clave. Son las siguientes:

| Operador | Tipo de datos de XPath |
|---------------------|-------------------------------|
| <code>&=</code> | <code>and_eq</code> |
| <code> =</code> | <code>or_eq</code> |
| <code>^=</code> | <code>xor_eq</code> |

C++ especifica estas palabras clave de operador como ortografía alternativa para los operadores de asignación compuesta. En C, las ortografías alternativas se proporcionan como macros en el encabezado de `<iso646.h>`. En C++, las ortografías alternativas son palabras clave; el uso de `<iso646.h>` o el equivalente C++ `<ciso646>` está en desuso. En Microsoft C++, se requiere la opción del compilador `/permissive-` o `/Za` para poder habilitar la ortografía alternativa.

Ejemplo

C++

```

// expre_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;
}

```

Asignación simple

El operador de asignación simple (`=`) hace que el valor del segundo operando se almacene en el objeto especificado por el primer operando. Si ambos objetos son de tipos aritméticos, el operando derecho se convierte al tipo de la izquierda antes de almacenar el valor.

Los objetos de `const` y `volatile` se pueden asignar a valores L de tipos que solo son `volatile`, o que no son `const` o `volatile`.

La asignación a objetos de tipo de clase (`struct`, `union` y `class`) se realiza mediante una función denominada `operator=`. El comportamiento predeterminado de esta función de operador es realizar una copia bit a bit; sin embargo, este comportamiento se puede modificar con operadores sobrecargados. Para obtener más información, vea [Sobrecarga de operadores](#). Los tipos de clase también pueden tener operadores de *asignación de copia* y *asignación de movimiento*. Para obtener más información, consulte [Copiar constructores y copiar operadores de asignación](#) y [Mover constructores y mover operadores de asignación](#).

Un objeto de cualquier clase derivada sin ambigüedad de una clase base se puede asignar a un objeto de la clase base. Lo contrario no es cierto porque hay una conversión implícita de la clase derivada a la clase base, pero no de la clase base a la clase derivada. Por ejemplo:

```

// 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
}

```

Las asignaciones a los tipos de referencia se comportan como si la asignación se creara en el objeto al que señala la referencia.

Para los objetos de tipo de clase, la asignación es diferente de la inicialización. Para ver lo diferentes que pueden ser la asignación y la inicialización, considere el código

C++

```

UserType1 A;
UserType2 B = A;

```

El código anterior muestra un inicializador; llama al constructor para `UserType2` que toma un argumento de tipo `UserType1`. Dado el código

C++

```

UserType1 A;
UserType2 B;

B = A;

```

la instrucción de asignación

C++

```
B = A;
```

puede tener uno de los efectos siguientes:

- Llame a la función `operator=` para `UserType2`, siempre que `operator=` tenga un argumento `UserType1`.
- Llamar a la función de conversión explícita `UserType1::operator UserType2`, si existe tal función.
- Llamar a un constructor `UserType2::UserType2`, siempre que exista un constructor de ese tipo, que tome un argumento `UserType1` y copie el resultado.

Asignación compuesta

Los operadores de asignación compuesta se muestran en la tabla [Operadores de asignación](#). Estos operadores tienen el formato $e1op= e2$, donde $e1$ es un valor L modifiable que no `const` es y $e2$ es:

- un tipo aritmético
- un puntero, si op es `+` o `-`

El $e1op= e2$ se comporta como $e1 = e1ope2$, pero $e1$ solo se evalúa una vez.

La asignación compuesta a un tipo enumerado genera un mensaje de error. Si el operando izquierdo tiene el tipo de puntero, el operando derecho debe tener el tipo de puntero o debe ser una expresión de constante que se evalúa como 0. Cuando el operando izquierdo es de tipo integral, el operando derecho no debe ser de tipo apuntador.

Resultado de los operadores de asignación

Los operadores de asignación devuelven el valor del objeto especificado por el operando izquierdo después de la asignación. El tipo resultante es el tipo del operando izquierdo. El resultado de una expresión de asignación es siempre un valor L. Estos operadores tienen asociatividad de derecha a izquierda. El operando izquierdo debe ser un valor L modifiable.

En ANSI C, el resultado de una expresión de asignación no es un valor L. Esto significa que la expresión de C++ `(a += b) += c` no se permite en C.

Consulte también

[Expresiones con operadores binarios](#)

[Operadores integrados de C++, precedencia y asociatividad](#)

[Operadores de asignación de C](#)

Operador AND bit a bit: &

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Sintaxis

and-expression:

equality-expression

and-expression & *equality-expression*

Comentarios

El operador AND bit a bit (&) compara cada bit del primer operando con el bit correspondiente del segundo operando. Si ambos bits son 1, el bit del resultado correspondiente se establece en 1. De lo contrario, el bit del resultado correspondiente se establece en 0.

Ambos operandos para el operador AND bit a bit deben tener tipos enteros. Las conversiones aritméticas habituales descritas en [Conversiones estándar](#) se aplican a los operandos.

Palabra clave del operador para &

C++ especifica `bitand` como una ortografía alternativa para &. En C, la ortografía alternativa se proporciona como una macro en el encabezado `<iso646.h>`. En C++, la ortografía alternativa es una palabra clave; el uso de `<iso646.h>` o el equivalente de C++ `<ciso646>` está en desuso. En Microsoft C++, se requiere la opción del compilador `/permissive-` o `/Za` para poder habilitar la ortografía alternativa.

Ejemplo

C++

```
// expre_Bitwise_AND_Operator.cpp
// compile with: /EHsc
// Demonstrate bitwise AND
#include <iostream>
using namespace std;
int main() {
    unsigned short a = 0CCCC;           // pattern 1100 ...
    unsigned short b = 0xAAAA;          // pattern 1010 ...
```

```
    cout << hex << ( a & b ) << endl; // prints "8888", pattern 1000 ...
}
```

Consulte también

[Operadores integrados de C++, precedencia y asociatividad](#)

[Operadores bit a bit de C](#)

Operador OR exclusivo bit a bit: ^

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Sintaxis

`expression ^ expression`

Comentarios

El operador OR exclusivo bit a bit (`^`) compara cada bit de su primer operando con el bit correspondiente de su segundo operando. Si el bit de uno de los operandos es 0 y el bit del otro operando es 1, el bit del resultado correspondiente se establece en 1. De lo contrario, el bit del resultado correspondiente se establece en 0.

Ambos operandos para el operador deben tener tipos enteros. Las conversiones aritméticas habituales descritas en [Conversiones aritméticas](#) se aplican a los operandos.

Para obtener más información sobre el uso alternativo del carácter `^` en C++/CLI y C++/CX, consulte [Identificador a un operador de objeto \(^\) \(C++/CLI y C++/CX\)](#).

Palabra clave del operador para ^

C++ especifica `xor` como una ortografía alternativa para `^`. En C, la ortografía alternativa se proporciona como una macro en el encabezado `<iso646.h>`. En C++, la ortografía alternativa es una palabra clave; el uso de `<iso646.h>` o el equivalente de C++ `<ciso646>` está en desuso. En Microsoft C++, se requiere la opción del compilador `/permissive-` o `/Za` para poder habilitar la ortografía alternativa.

Ejemplo

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 ...  
}
```

Consulte también

[Operadores integrados de C++, precedencia y asociatividad](#)

Operador OR inclusivo bit a bit: |

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Sintaxis

`expression1 | expression2`

Comentarios

El operador OR inclusivo bit a bit (|) compara cada bit de su primer operando con el bit correspondiente de su segundo operando. Si uno de los dos bits es 1, el bit del resultado correspondiente se establece en 1. De lo contrario, el bit del resultado correspondiente se establece en 0.

Ambos operandos para el operador deben tener tipos enteros. Las conversiones aritméticas habituales descritas en [Conversiones aritméticas](#) se aplican a los operandos.

Palabra clave del operador para |

C++ especifica `bitor` como una ortografía alternativa para |. En C, la ortografía alternativa se proporciona como una macro en el encabezado `<iso646.h>`. En C++, la ortografía alternativa es una palabra clave; el uso de `<iso646.h>` o el equivalente de C++ `<ciso646>` está en desuso. En Microsoft C++, se requiere la opción del compilador `/permissive-` o `/Za` para poder habilitar la ortografía alternativa.

Ejemplo

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 1010 ...

    cout << hex << ( a | b ) << endl; // prints "ffff" pattern 1111 ...
}
```

Consulte también

[Operadores integrados de C++, precedencia y asociatividad](#)

[Operadores bit a bit de C](#)

Operador de conversión: ()

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Una conversión de tipo proporciona un método para la conversión explícita del tipo de un objeto en una situación concreta.

Sintaxis

```
cast-expression:  
    unary-expression  
    ( type-name ) cast-expression
```

Comentarios

Cualquier expresión unaria se considera una expresión de conversión.

El compilador trata `cast-expression` como tipo `type-name` una vez realizada una conversión de tipo. Las conversiones se pueden utilizar para convertir objetos de cualquier tipo escalar en cualquier otro tipo escalar. Las conversiones de tipos explícitas están restringidas por las mismas reglas que determinan los efectos de las conversiones implícitas. Otras restricciones sobre las conversiones pueden deberse a los tamaños reales o de la representación de tipos específicos.

Ejemplos

Una conversión estándar entre tipos integrados:

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;
}
```

Un operador de conversión definido en un tipo definido por el usuario:

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
```

Consulte también

[Expresiones con operadores unarios](#)

[Operadores integrados de C++, precedencia y asociatividad](#)

[Operador de conversión explícita de tipos: \(\)](#)

[Operadores de conversión \(C++\)](#)

[Operadores de conversión \(C\)](#)

Operador coma: ,

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Permite agrupar dos instrucciones donde se espera una.

Sintaxis

```
expression , expression
```

Comentarios

El operador de comas tiene asociatividad de izquierda a derecha. Dos expresiones separadas por una coma se evalúan de izquierda a derecha. El operando izquierdo se evalúa siempre, y todos los efectos secundarios se completan antes de que se evalúe el operando derecho.

Las comas se pueden utilizar como separadores en algunos contextos, como las listas de argumentos de función. No confunda el uso de la coma como separador y como operador; los dos usos son completamente diferentes.

Considere la expresión `e1, e2`. El tipo y el valor de la expresión son el tipo y el valor de `e2`; el resultado de evaluar `e1` se descarta. El resultado es un valor L si el operando derecho es un valor L.

En los contextos en los que normalmente se utiliza la coma como separador (por ejemplo, en argumentos reales para funciones o inicializadores de agregado), el operador de la coma y sus operandos deben incluirse entre paréntesis. Por ejemplo:

C++

```
func_one( x, y + 2, z );
func_two( (x--, y + 2), z );
```

En la llamada de función a `func_one` de la parte superior, se pasan tres argumentos separados por comas: `x`, `y + 2` y `z`. En la llamada de función a `func_two`, los paréntesis hacen que el compilador interprete la primera coma como el operador de evaluación secuencial. Esta llamada a función pasa dos argumentos a `func_two`. El primer

argumento es el resultado de la operación de evaluación secuencial `(x--, y + 2)`, que tiene el valor y tipo de la expresión `y + 2`; el segundo argumento es `z`.

Ejemplo

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
```

Consulte también

[Expresiones con operadores binarios](#)

[Operadores integrados de C++, precedencia y asociatividad](#)

[Operador de evaluación secuencial](#)

Operador condicional ?:

Artículo • 26/09/2022 • Tiempo de lectura: 2 minutos

Sintaxis

```
expression ? expression : expression
```

Comentarios

El operador condicional (?:) es un operador ternario (es decir, toma tres operandos). El operador condicional funciona del modo siguiente:

- El primer operando se convierte implícitamente a `bool`. Se evalúa y todos los efectos secundarios se completan antes de continuar.
- Si el primer operando se evalúa como `true` (1), se evalúa el segundo operando.
- Si el primer operando se evalúa como `false` (0), se evalúa el tercer operando.

El resultado del operador condicional es el resultado de cualquier operando que se evalúe, el segundo o el tercero. Solo uno de los dos últimos operandos se evalúa en una expresión condicional.

Las expresiones condicionales tienen asociatividad de derecha a izquierda. El primer operando debe ser de tipo entero o de tipo de puntero. Las reglas siguientes se aplican a los operandos segundo y tercero:

- Si ambos operandos son del mismo tipo, el resultado es de ese tipo.
- Si ambos operandos son de tipos de aritmética o de enumeración, se realizan las conversiones aritméticas habituales (que se describen en [Conversiones estándar](#)) para convertirlos a un tipo común.
- Si ambos operandos son de tipos de puntero o si uno es de un tipo de puntero y el otro es una expresión de constante que se evalúa como 0, las conversiones de puntero se realizan para convertirlos a un tipo común.
- Si ambos operandos son de tipos de referencia, las conversiones de referencia se realizan para convertirlos a un tipo común.

- Si ambos operandos son de tipo void, el tipo común es el tipo void.
- Si ambos operandos son del mismo tipo definido por el usuario, el tipo común es ese tipo.
- Si los operandos son de tipos diferentes y al menos uno de ellos tiene un tipo definido por el usuario, se usan reglas de lenguaje para determinar el tipo común. (Vea la advertencia más adelante).

Las combinaciones de los operandos segundo y tercero no incluidos en la lista anterior no son válidas. El tipo del resultado es el tipo común, y es un valor L si tanto el segundo como el tercer operando son del mismo tipo y ambos son valores I.

Advertencia

Si los tipos de los operandos segundo y tercero no son idénticos, se invocan reglas de conversión de tipo complejo, como se especifica en el estándar de C++. Estas conversiones pueden provocar un comportamiento inesperado, incluida la creación y destrucción de objetos temporales. Por esta razón, recomendamos encarecidamente (1) evitar el uso de tipos definidos por el usuario como operandos con el operador condicional, o (2) si utiliza tipos definidos por el usuario, convertir explícitamente cada operando a un tipo común.

Ejemplo

C++

```
// expre_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;
}
```

Consulte también

[Operadores integrados de C++, precedencia y asociatividad](#)

[Operador de expresión condicional](#)

delete (Operador) (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Desasigna un bloque de memoria.

Sintaxis

```
[::] delete cast-expression  
[::] delete [] cast-expression
```

Comentarios

El argumento *cast-expression* debe ser un puntero a un bloque de memoria asignado previamente para un objeto creado con el [operador new](#). El operador `delete` tiene un resultado de tipo `void` y, por tanto, no devuelve ningún valor. Por ejemplo:

C++

```
CDialog* MyDialog = new CDialog;  
// use MyDialog  
delete MyDialog;
```

El uso de `delete` en un puntero a un objeto no asignado con `new` tiene resultados impredecibles. Sin embargo, se puede usar `delete` en un puntero con el valor 0. Esta disposición significa que, cuando `new` devuelve 0 en caso de error, la eliminación del resultado de una operación `new` errónea es inofensiva. Para más información, consulte [Los operadores "new" y "delete"](#).

Los operadores `new` y `delete` también se pueden usar con tipos integrados, incluidas las matrices. Si `pointer` hace referencia a una matriz, ponga corchetes vacíos (`[]`) delante de `pointer`:

C++

```
int* set = new int[100];  
//use set[]  
delete [] set;
```

El uso del operador `delete` en un objeto desasigna su memoria. Un programa que desreferencia un puntero después de eliminarse el objeto puede producir resultados

impredecibles o bloquearse.

Cuando se usa `delete` para desasignar memoria para un objeto de clase de C++, se llama al destructor del objeto antes de que se desasigne la memoria del objeto (si el objeto tiene un destructor).

Si el operando del operador `delete` es un valor l modifiable, su valor es indefinido después de que se elimina el objeto.

Si se especifica la opción del compilador [/sdl \(Habilitar comprobaciones de seguridad adicionales\)](#), el operando al operador `delete` se establece en un valor no válido después de eliminar el objeto.

Usar `delete`

Hay dos variantes sintácticas para el [operador `delete`](#): una para objetos únicos y la otra para matrices de objetos. En el fragmento de código siguiente se muestran las diferencias entre ellas:

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;
}
```

Los dos casos siguientes generan resultados indefinidos: usando la forma de matriz de `delete` (`delete []`) en un objeto y usando la forma no de matriz de `delete` en una matriz.

Ejemplo

Para ver ejemplos de uso de `delete`, consulte [operador new](#).

Cómo eliminar trabajos

El operador `delete` invoca la función [operator delete](#).

Para los objetos que no son de tipo de clase ([clase](#), [struct](#) o [unión](#)), se invoca el operador `delete` global. Para los objetos de tipo de clase, el nombre de la función de desasignación se resuelve en el ámbito global si la expresión de eliminación comienza con el operador unario de resolución de ámbito (`::`). Si no, el operador `delete` invoca el destructor de un objeto antes de la desasignación de memoria (si el puntero no es nulo). El operador `delete` puede definirse clase por clase; si no existe esta definición para una clase determinada, se invoca el operador `delete` global. Si la expresión de eliminación se utiliza para desasignar un objeto de clase cuyo de tipo estático tenga un destructor virtual, la función de desasignación se resuelve mediante el destructor virtual del tipo dinámico del objeto.

Vea también

[Expresiones con operadores unarios](#)

[Palabras clave](#)

[Operadores new y delete](#)

Operadores de igualdad: == y !=

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Sintaxis

```
expression == expression  
expression != expression
```

Comentarios

Los operadores de igualdad binarios comparan la igualdad o desigualdad estricta de sus operandos.

Los operadores de igualdad, igual a (==) y no igual a (!=), tienen prioridad inferior que los operadores relacionales, pero se comportan de igual forma. El tipo de resultado de estos operadores es `bool`.

El operador igual a (==) devuelve `true` si ambos operandos tienen el mismo valor; de lo contrario, devuelve `false`. El operador no es igual a (!=) devuelve `true` si los operandos no tienen el mismo valor; de lo contrario, devuelve `false`.

Palabra clave del operador para !=

C++ especifica `not_eq` como una ortografía alternativa para !=. (No hay ninguna ortografía alternativa para ==). En C, la ortografía alternativa se proporciona como una macro en el encabezado <iso646.h>. En C++, la ortografía alternativa es una palabra clave; el uso de <iso646.h> o el equivalente de C++ <ciso646> está en desuso. En Microsoft C++, se requiere la opción del compilador [/permissive-](#) o [/Za](#) para poder habilitar la ortografía alternativa.

Ejemplo

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

```
int main() {
    cout << boolalpha
        << "The true expression 3 != 2 yields: "
        << (3 != 2) << endl
        << "The false expression 20 == 10 yields: "
        << (20 == 10) << endl;
}
```

Los operadores de igualdad puede comparar punteros a miembros del mismo tipo. En esta comparación, se realizan conversiones de puntero a miembro. Los punteros a miembros también se pueden comparar con una expresión constante que se evalúa como 0.

Consulte también

[Expresiones con operadores binarios](#)

[Operadores integrados de C++, precedencia y asociatividad](#)

[Operadores relacionales y de igualdad de C](#)

Operador de conversión explícita de tipos: ()

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

C++ permite la conversión de tipos explícita mediante una sintaxis similar a la de las llamadas de función.

Sintaxis

```
simple-type-name ( expression-list )
```

Comentarios

Un elemento *simple-type-name* seguido de una *expression-list* incluida entre paréntesis crea un objeto del tipo indicado mediante las expresiones especificadas. En el ejemplo siguiente se muestra una conversión de tipo explícita al tipo int:

```
C++
```

```
int i = int( d );
```

En el ejemplo siguiente, se muestra una clase `Point`.

Ejemplo

```
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();
}

```

Resultados

Output

```
x = 20, y = 10
x = 0, y = 0
```

Aunque el ejemplo anterior muestra la conversión de tipos explícita mediante constantes, se puede usar la misma técnica para realizar estas conversiones en objetos. En el siguiente fragmento de código se muestra este caso:

C++

```

int i = 7;
float d;

d = float( i );

```

Las conversiones de tipos explícitas también se pueden especificar utilizando la sintaxis de conversión ("cast"). El ejemplo anterior, reescrito mediante la sintaxis de conversión, es:

C++

```
d = (float)i;
```

Las conversiones de estilo "cast" o de función tienen los mismos resultados cuando se convierten valores simples. Sin embargo, en la sintaxis de estilo de función, puede especificar más de un argumento para la conversión. Esta diferencia es importante para los tipos definidos por el usuario. Considere una clase `Point` y sus conversiones:

C++

```
struct Point
{
    Point( short x, short y ) { _x = x; _y = y; }
    ...
    short _x, _y;
};
...
Point pt = Point( 3, 10 );
```

En el ejemplo anterior, donde se usa la conversión de estilo de función, se muestra cómo convertir dos valores (uno para `x` y uno para `y`) al tipo `Point` definido por el usuario.

⊗ Precaución

Utilice las conversiones de tipos explícitas con cuidado, ya que invalidan la comprobación de tipos integrada del compilador de C++.

La notación de conversión `cast` se debe usar para las conversiones a tipos que no tienen un elemento *simple-type-name* (por ejemplo, tipos de puntero o referencia). La conversión a tipos que puedan expresarse con un elemento *simple-type-name* se puede escribir en cualquiera de las formas.

La definición de tipos en las conversiones "cast" no es válida.

Consulte también

Expresiones postfijas

Operadores integrados de C++, precedencia y asociatividad

Operador de llamada de función: ()

Artículo • 03/03/2023 • Tiempo de lectura: 4 minutos

Una llamada de función es un tipo de *postfix-expression* formada por una expresión que se evalúa como una función o un objeto al que se puede llamar seguido del operador de llamada de función, `()`. Un objeto puede declarar una función *operator ()*, lo que proporciona semántica de llamada de función para el objeto.

Sintaxis

postfix-expression:

postfix-expression `(` *argument-expression-list* `opt` `)`

Comentarios

Los argumentos del operador de llamada de función proceden de *argument-expression-list*, una lista de expresiones separadas por comas. Los valores de estas expresiones se pasan a la función como argumentos. El elemento *argument-expression-list* puede estar vacío. Antes de C++ 17, el orden de evaluación de la expresión de función y las expresiones de argumento no se especificaba y podía ejecutarse en cualquier orden. En C++17 y versiones posteriores, la expresión de función se evalúa antes que cualquier expresión de argumento o argumento predeterminado. Las expresiones de argumento se evalúan en una secuencia indeterminada.

postfix-expression se evalúa como la función a la que se va a llamar. Puede tomar cualquiera de entre varias formas:

- un identificador de función, visible en el ámbito actual o en el ámbito de cualquiera de los argumentos de función proporcionados,
- una expresión que se evalúa como una función, un puntero de función, un objeto al que se puede llamar o una referencia a uno,
- un descriptor de acceso de función miembro, ya sea explícito o implícito,
- un puntero desreferenciado a una función miembro.

postfix-expression puede ser un identificador de función sobrecargado o un descriptor de acceso de función miembro sobrecargado. Las reglas para la resolución de sobrecargas determinan la función real a la que se va a llamar. Si la función miembro es virtual, la función a la que se va a llamar se determina en tiempo de ejecución.

Algunas declaraciones de ejemplo:

- Una función que devuelve el tipo `T`. Una declaración de ejemplo es

C++

```
T func( int i );
```

- Un puntero a una función que devuelve el tipo `T`. Una declaración de ejemplo es

C++

```
T (*func)( int i );
```

- Una referencia a una función que devuelve el tipo `T`. Una declaración de ejemplo es

C++

```
T (&func)(int i);
```

- Una desreferencia de función de puntero a miembro que devuelve el tipo `T`. Estos son algunos ejemplos de llamadas de función:

C++

```
(pObject->*pmf)();  
(Object.*pmf)();
```

Ejemplo

En el ejemplo siguiente se llama a la función de biblioteca estándar `strcat_s` con tres argumentos:

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++
```

Resultados de la llamada de función

Una llamada de función se evalúa como rvalue a menos que la función se declare como un tipo de referencia. Las funciones con tipos devueltos de referencia se evalúan como lvalues. Estas funciones se pueden usar en el lado izquierdo de una instrucción de asignación, como se muestra aquí:

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";
}

```

El código anterior define una clase de nombre `Point` que contiene objetos de datos privados que representan coordenadas `x` e `y`. Estos objetos de datos se deben modificar y sus valores se deben recuperar. Este programa es solo uno de varios diseños para esa clase; el uso de las funciones `GetX` y `SetX` o `GetY` y `SetY` es otro diseño posible.

Las funciones que devuelven tipos de clase, punteros a tipos de clase o referencias a tipos de clase se pueden utilizar como operando izquierdo para operadores de selección de miembros. El código siguiente es legal:

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 );
}
```

Las funciones se pueden llamar de forma recursiva. Para obtener más información declaraciones de función, vea [Funciones](#). Hay material relacionado en [Unidades de traducción y vinculación](#).

Consulte también

[Expresiones de postfijo](#)

[Operadores integrados de C++, precedencia y asociatividad](#)

[Llamada de función](#)

Operador de direccionamiento indirecto: *

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Sintaxis

```
* cast-expression
```

Comentarios

El operador de direccionamiento indirecto unario (*) desreferencia un puntero; es decir, convierte un valor de puntero en un valor L. El operando del operador de direccionamiento indirecto debe ser un puntero a un tipo. El resultado de la expresión de direccionamiento indirecto es el tipo del que se deriva el tipo de puntero. El uso del operador * en este contexto es diferente de su significado como operador binario, que es multiplicación.

Si el operando señala a una función, el resultado es un designador de función. Si señala a una ubicación de almacenamiento, el resultado es un valor L que designa la ubicación de almacenamiento.

El operador de direccionamiento indirecto se puede utilizar de manera acumulativa para desreferenciar punteros a punteros. Por ejemplo:

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;  
}
```

Si el valor de puntero no es válido, el resultado es indefinido. La lista siguiente incluye algunas de las condiciones más comunes que invalidan un valor de puntero.

- El puntero es un puntero null.
- El puntero especifica la dirección de un elemento local que no está visible en el momento de la referencia.
- El puntero especifica una dirección que está alineada de una forma no adecuada para el tipo de objeto al que se señala.
- El puntero especifica una dirección no utilizada por el programa que se ejecuta.

Vea también

[Expresiones con operadores unarios](#)

[Operadores integrados de C++, precedencia y asociatividad](#)

[Operador address-of: &](#)

[Operadores de direccionamiento indirecto y address-of](#)

Operadores de desplazamiento a la izquierda y a la derecha (<< y >>)

Artículo • 14/03/2023 • Tiempo de lectura: 6 minutos

Los operadores de desplazamiento bit a bit son el operador de desplazamiento a la derecha (`>>`), que mueve los bits de una expresión de tipo entero o de enumeración a la derecha, y el operador de desplazamiento izquierdo (`<<`), que mueve los bits a la izquierda.¹

Sintaxis

```
shift-expression:  
    additive-expression  
    shift-expression << additive-expression  
    shift-expression >> additive-expression
```

Comentarios

ⓘ Importante

Las descripciones y los ejemplos siguientes son válidos en Windows para las arquitecturas x86 y x64. La implementación de los operadores de desplazamiento a la izquierda y desplazamiento a la derecha es muy diferente en los dispositivos Windows para ARM. Para obtener más información, vea la sección sobre los operadores de desplazamiento en la entrada de blog [Hello ARM ↗](#).

Desplazamientos a la izquierda

El operador de desplazamiento a la izquierda hace que los bits de `shift-expression` se desplacen hacia la izquierda el número de posiciones especificado por `additive-expression`. Las posiciones de bits que quedan vacantes debido a la operación de desplazamiento se rellenan con ceros. Un desplazamiento a la izquierda es un desplazamiento lógico (los bits que se desplazan más allá del final se descartan, incluido el bit de signo). Para obtener más información sobre los tipos de desplazamiento bit a bit, vea el artículos sobre los [desplazamientos bit a bit ↗](#).

En el ejemplo siguiente se muestran operaciones de desplazamiento a la izquierda con números sin signo. El ejemplo muestra lo que ocurre con los bits representando el valor como unbitset. Para obtener más información, consulte [Clase bitset](#).

C++

```
#include <iostream>
#include <bitset>

using namespace std;

int main() {
    unsigned short short1 = 4;
    bitset<16> bitset1{short1};    // the bitset representation of 4
    cout << bitset1 << endl;    // 0b00000000'00000100

    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
}
```

Si desplaza a la izquierda un número con signo de modo que el bit de signo se vea afectado, el resultado es indefinido. En el ejemplo siguiente se muestra lo que ocurre cuando un 1 bit se desplaza a la izquierda hasta la posición del bit de signo.

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
}
```

Desplazamientos a la derecha

El operador de desplazamiento a la derecha hace que el patrón de bits de `shift-expression` se desplace hacia la derecha el número de posiciones especificado por `additive-expression`. En el caso de números sin signo, las posiciones de bits que quedan vacantes debido a la operación de desplazamiento se rellenan con ceros. Si se trata de números con signo, el bit de signo se emplea para llenar las posiciones de bits vacantes. Es decir, si el número es positivo se usa 0 y si el número es negativo se usa 1.

ⓘ Importante

El resultado de un desplazamiento a la derecha de un número negativo con signo depende de la implementación. Aunque el compilador de Microsoft C++ usa el bit de signo para llenar las posiciones de bits vacantes, no hay ninguna garantía de que otras implementaciones también lo hagan.

En este ejemplo se muestran operaciones de desplazamiento a la derecha que usan números sin signo:

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
}
```

En el ejemplo siguiente se muestran operaciones de desplazamiento a la derecha con números positivos con signo.

```
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  
}
```

En el ejemplo siguiente se muestran operaciones de desplazamiento a la derecha con enteros negativos con signo.

```
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
}
```

Desplazamientos y promociones

Las expresiones especificadas a ambos lados de un operador de desplazamiento deben ser de tipos enteros. Las promociones de enteros se realizan según las reglas que se describen en [Conversiones estándar](#). El tipo del resultado es el mismo que el tipo de *shift-expression* que se ha promovido.

En el ejemplo siguiente, una variable de tipo `char` se promueve a `int`.

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
}
```

Detalles

El resultado de una operación de desplazamiento es indefinido si *additive-expression* es negativo o si *additive-expression* es mayor o igual que el número de bits de *shift-expression* (que se ha promovido). No se realiza ninguna operación de desplazamiento si *additive-expression* es 0.

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)
}

```

Notas al pie

¹ A continuación se muestra la descripción de los operadores de desplazamiento en la especificación ISO de C++ 11 (INCITS/ISO/IEC 14882-2011[2012]), secciones 5.8.2 y 5.8.3.

El valor de `E1 << E2` es `E1` desplazado a la izquierda `E2` posiciones de bits; los bits vacantes se rellenan con ceros. Si `E1` tiene un tipo sin signo, el valor resultante es $E1 \times 2^{E2}$, reduciendo cada módulo en uno más que el valor máximo que se puede representar en el tipo del resultado. De lo contrario, si `E1` tiene un tipo con signo y un valor no negativo, y $E1 \times 2^{E2}$ se puede representar en el tipo sin signo correspondiente del tipo del resultado, ese valor, convertido al tipo del resultado, es el valor resultante; de lo contrario, el comportamiento es indefinido.

El valor de `E1 >> E2` es `E1` desplazado a la derecha `E2` posiciones de bits. Si `E1` tiene un tipo sin signo o si `E1` tiene un tipo con signo y un valor no negativo, el valor del resultado es la parte entera del cociente de $E1/2^{E2}$. Si `E1` tiene un tipo con signo y un valor negativo, el valor resultante está definido por la implementación.

Consulte también

[Expresiones con operadores binarios](#)

[Operadores integrados de C++, precedencia y asociatividad](#)

Operador AND lógico: `&&`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Sintaxis

Logical-and-expression:

equality-expression

logical-and-expression `&&` *equality-expression*

Comentarios

El operador AND lógico (`&&`) devuelve `true` si los dos operandos son `true`; de lo contrario, devuelve `false`. Los operandos se convierten implícitamente al tipo `bool` antes de la evaluación y el resultado es de tipo `bool`. El operador AND lógico tiene asociatividad de izquierda a derecha.

Los operandos del operador lógico AND no necesitan tener el mismo tipo, pero deben ser de tipo booleano, entero o de puntero. Los operandos son normalmente expresiones relacionales o de igualdad.

El primer operando se evalúa en su totalidad y, antes de proseguir con la evaluación de la expresión AND lógica, se aplican todos los efectos secundarios.

El segundo operando se evalúa solo si el primero se evalúa como `true` (distinto de cero). Esta evaluación elimina la evaluación innecesaria del segundo operando cuando la expresión lógica AND es `false`. Puede utilizar esta evaluación de cortocircuito para evitar la desreferencia de punteros null, como se muestra en el ejemplo siguiente:

C++

```
char *pch = 0;  
// ...  
(pch) && (*pch = 'a');
```

Si `pch` es null (0), el lado derecho de la expresión no se evalúa. Esta evaluación de cortocircuito hace imposible la asignación a través de un puntero nulo.

Palabra clave del operador para `&&`

C++ especifica `and` como una ortografía alternativa para `&&`. En C, la ortografía alternativa se proporciona como una macro en el encabezado `<iso646.h>`. En C++, la ortografía alternativa es una palabra clave; el uso de `<iso646.h>` o el equivalente de C++ `<ciso646>` está en desuso. En Microsoft C++, se requiere la opción del compilador `/permissive-` o `/Za` para poder habilitar la ortografía alternativa.

Ejemplo

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;
}
```

Consulte también

[Operadores integrados de C++, precedencia y asociatividad](#)
[Operadores lógicos de C](#)

Operador de negación lógico: !

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Sintaxis

`!cast-expression`

Comentarios

El operador de negación lógico (`!`) invierte el significado del operando. El operando debe ser de tipo aritmético o de puntero (o una expresión que se evalúe como un tipo aritmético o de puntero). El operando se convierte implícitamente al tipo `bool`. El resultado es `true` si el operando convertido es `false`; el resultado es `false` si el operando convertido es `true`. El resultado es de tipo `bool`.

Para una expresión `e`, la expresión unaria `!e` es equivalente a la expresión `(e == 0)`, excepto si intervienen operadores sobrecargados.

Palabra clave del operador para !

C++ especifica `not` como una ortografía alternativa para `!`. En C, la ortografía alternativa se proporciona como una macro en el encabezado `<iso646.h>`. En C++, la ortografía alternativa es una palabra clave; el uso de `<iso646.h>` o el equivalente de C++ `<ciso646>` está en desuso. En Microsoft C++, se requiere la opción del compilador `/permissive-` o `/Za` para poder habilitar la ortografía alternativa.

Ejemplo

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;
}
```

Consulte también

[Expresiones con operadores unarios](#)

[Operadores integrados de C++, precedencia y asociatividad](#)

[Operadores aritméticos unarios](#)

Operador lógico OR: ||

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Sintaxis

logical-or-expression || *logical-and-expression*

Comentarios

El operador lógico OR (||) devuelve el valor booleano `true` si uno o ambos operandos son `true` y, de lo contrario, devuelven `false`. Los operandos se convierten implícitamente al tipo `bool` antes de la evaluación y el resultado es de tipo `bool`. El operador OR lógico tiene asociatividad de izquierda a derecha.

Los operandos del operador lógico OR no tienen que tener el mismo tipo, pero deben ser de tipo booleano, entero o de puntero. Los operandos son normalmente expresiones relacionales o de igualdad.

El primer operando se evalúa en su totalidad y, antes de proseguir con la evaluación de la expresión OR lógica, se aplican todos los efectos secundarios.

El segundo operando se evalúa solo si el primer operando se evalúa como `false`, porque la evaluación no es necesaria cuando la expresión lógica OR es `true`. Se conoce como evaluación de *cortocircuito*.

C++

```
printf( "%d" , (x == w || x == y || x == z) );
```

En el ejemplo anterior, si `x` es igual a `w`, `y` o `z`, el segundo argumento de la función `printf` se evalúa como `true`, que luego se promueve a un número entero, y se imprime el valor 1. De lo contrario, se evalúa como `false` y se imprime el valor 0. En cuanto una de las condiciones se evalúe como `true`, se interrumpe la evaluación.

Palabra clave del operador para ||

C++ especifica `or` como una ortografía alternativa para `||`. En C, la ortografía alternativa se proporciona como una macro en el encabezado `<iso646.h>`. En C++, la ortografía alternativa es una palabra clave; el uso de `<iso646.h>` o el equivalente de

C++ <ciso646> está en desuso. En Microsoft C++, se requiere la opción del compilador [/permissive-](#) o [/Za](#) para poder habilitar la ortografía alternativa.

Ejemplo

```
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;  
}
```

Consulte también

[Operadores integrados de C++, precedencia y asociatividad](#)

[Operadores lógicos de C](#)

Operadores de acceso a miembros: . y ->

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Sintaxis

postfix-expression:

```
postfix-expression . templateopt id-expression  
postfix-expression -> templateopt id-expression
```

Comentarios

Los operadores de acceso a miembros . y -> se usan para hacer referencia a los miembros de los tipos `struct`, `union` y `class`. Las expresiones de acceso a miembros tienen el valor y el tipo del miembro seleccionado.

Hay dos formas de expresiones de acceso de miembro:

1. En la primera forma, `postfix-expression` representa un valor de tipo `struct`, `class` o `union`, y `id-expression` nombra a un miembro especificado `struct`, `union` o `class`. El valor de la operación es el de `id-expression` y es un valor L si `postfix-expression` es un valor L.
2. En el segundo formulario, `postfix-expression` representa un puntero a `struct`, `union` o `class`, y `id-expression` nombra a un miembro especificado `struct`, `union` o `class`. El valor es el de `id-expression` y es un valor L. El operador `->` desreferencia el puntero. Las expresiones `e->member` y `(*(e)).member` (donde `e` representa un puntero) producen resultados idénticos (excepto cuando se sobrecargan los operadores `->` o `*`).

Ejemplo

En el ejemplo siguiente se muestran dos formas del operador de acceso a miembros.

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
```

Consulte también

[Expresiones de postfijo](#)

[Operadores integrados de C++, precedencia y asociatividad](#)

[Clases y structs](#)

[Miembros de estructura y de unión](#)

Operadores de multiplicación y el operador de módulo

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Sintaxis

```
expression * expression
expression / expression
expression % expression
```

Comentarios

Los operadores multiplicativos son:

- Multiplicación (*)
- División (/)
- Módulo (resto de la división) (%)

Estos operadores binarios tienen asociatividad de izquierda a derecha.

Los operadores multiplicativos toman operandos de tipos aritméticos. El operador de módulo (%) tiene un requisito más estricto en tanto que sus operandos deben ser de tipo entero. (Para obtener el resto de una división de punto flotante, use la función en tiempo de ejecución, [fmod](#)). Las conversiones que se tratan en [Conversiones estándar](#) se aplican a los operandos y el resultado es del tipo convertido.

El operador de multiplicación produce el resultado de multiplicar el primer operando por el segundo.

El operador de división produce el resultado de dividir el primer operando por el segundo.

El operador de módulo produce el resto proporcionado por la expresión siguiente, donde $e1$ es el primer operando y $e2$ es el segundo: $e1 - (e1 / e2) * e2$, donde ambos operandos son de tipos enteros.

La división por 0 en una expresión de división o de módulo es indefinida y provoca un error en tiempo de ejecución. Por consiguiente, las expresiones siguientes generan resultados erróneos, indefinidos:

C++

```
i % 0  
f / 0.0
```

Si ambos operandos de una multiplicación, una división o una expresión de módulo tienen el mismo signo, el resultado es positivo. De lo contrario, el resultado es negativo. El resultado del signo de una operación de módulo lo define la implementación.

① Nota

Como las conversiones realizadas por los operadores multiplicativos no proporcionan condiciones de desbordamiento o subdesbordamiento, la información puede perderse si el resultado de una operación multiplicativa no se puede representar en el tipo de los operandos después de la conversión.

Específicos de Microsoft

En Microsoft C++, el resultado de una expresión de módulo tiene siempre el mismo signo que el primer operando.

FIN de Específicos de Microsoft

Si la división calculada de dos enteros es inexacta y solo un operando es negativo, el resultado es el entero mayor (en magnitud, independientemente del signo) que es menor que el valor exacto que produciría la operación de división. Por ejemplo, el valor calculado de $-11 / 3$ es $-3,666666666$. El resultado de esa división entera es -3 .

La relación entre los operadores multiplicativos está determinada por la identidad $(e1 / e2) * e2 + e1 \% e2 == e1$.

Ejemplo

El siguiente programa muestra los operadores multiplicativos. Observe que ambos operandos de `10 / 3` se deben convertir explícitamente al tipo `float` para evitar el truncamiento, de manera que los dos operandos son de tipo `float` antes de la división.

C++

```
// expre_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;
}
```

Consulte también

[Expresiones con operadores binarios](#)

[Operadores integrados de C++, precedencia y asociatividad](#)

[Operadores de multiplicación de C](#)

new Operador (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 9 minutos

Intenta asignar e inicializar un objeto o matriz de objetos de un tipo especificado o de un marcador de posición, y devuelve un puntero con tipo adecuado y distinto de cero al objeto (o al objeto inicial de la matriz).

Sintaxis

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:

```
( expression-list opt )  
braced-init-list
```

Comentarios

Si no se ejecuta correctamente, `new` devuelve cero o arroja una excepción. Para más información, consulte [Los operadores new y delete](#). Puede cambiar este comportamiento predeterminado escribiendo una rutina de control de excepciones personalizada y llamando a la función de biblioteca en tiempo de ejecución `_set_new_handler` con el nombre de función como su argumento.

Para obtener más información sobre cómo crear un objeto en el montón administrado en C++/CLI y C++/CX, consulte [gcnew](#).

ⓘ Nota

Las extensiones de componentes de Microsoft C++ (C++/CX) proporcionan compatibilidad para que la palabra clave `new` agregue entradas de ranura de vtable. Para más información, consulte [new\(nueva ranura en vtable\)](#)

Cuando `new` se utiliza para asignar memoria para un objeto de clase de C++, se llama al constructor del objeto después de que se asigna la memoria.

Utilice el operador `delete` para desasignar la memoria asignada con el operador `new`. Use el operador `delete[]` para eliminar una matriz asignada por el operador `new`.

En el ejemplo siguiente se asigna y se libera una matriz bidimensional de caracteres de tamaño `dim` por 10. Al asignar una matriz multidimensional, todas las dimensiones excepto la primera deben ser expresiones constantes que se evalúen como valores positivos. La dimensión de matriz situada más a la izquierda puede ser cualquier expresión que se evalúe como un valor positivo. Al asignar una matriz mediante el operador `new`, la primera dimensión puede ser cero; el operador `new` devuelve un puntero único.

C++

```
char (*pchar)[10] = new char[dim][10];
delete [] pchar;
```

El elemento `type-id` no puede contener declaraciones de clase `const` o `volatile` ni declaraciones de enumeración. La siguiente expresión tiene un formato incorrecto:

C++

```
volatile char *vch = new volatile char[20];
```

El operador `new` no asigna tipos de referencia porque no son objetos.

El operador `new` no se puede usar para asignar una función, pero sí para asignar punteros a funciones. En el ejemplo siguiente se asigna y se libera una matriz de siete punteros a funciones que devuelven enteros.

C++

```
int (**p) () = new (int (*[7]) ());
delete p;
```

Si se usa el operador `new` sin ningún argumento adicional y compila con la opción `/GX`, `/EHs` o `/EHsc`, el compilador generará un código para llamar al operador `delete` si el constructor produce una excepción.

En la lista siguiente se describen los elementos de la gramática de `new`:

`new-placement`

Proporciona una manera de pasar argumentos extra si se sobrecarga `new`.

`type-id`

Especifica el tipo que se va a asignar; puede ser un tipo integrado o un tipo definido por el usuario. Si la especificación de tipo es compleja, puede ir entre paréntesis para forzar el orden de enlace. El tipo puede ser un marcador de posición (`auto`) cuyo tipo viene determinado por el compilador.

`new-initializer`

Proporciona un valor para el objeto inicializado. No se pueden especificar los inicializadores para las matrices. El operador `new` creará las matrices de objetos únicamente si la clase tiene un constructor predeterminado.

`noptr-new-declarator`

Especifica los límites de una matriz. Al asignar una matriz multidimensional, todas las dimensiones excepto la primera deben ser expresiones constantes que se evalúen como valores positivos convertibles a `std::size_t`. La dimensión de matriz situada más a la izquierda puede ser cualquier expresión que se evalúe como un valor positivo. El `attribute-specifier-seq` se aplica al tipo de matriz asociado.

Ejemplo: asignar y liberar una matriz de caracteres

En el ejemplo de código siguiente se asigna una matriz de caracteres y un objeto de clase `CName` y después se liberan.

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;
}

```

Por ejemplo: operador `new`

Si se usa el formato de ubicación del operador `new` (el formato con más argumentos que tamaño), el compilador no admite un formato de ubicación del operador `delete` si el constructor produce una excepción. Por ejemplo:

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;
}

```

Inicializar objetos asignados con `new`

Un campo `new-initializer` opcional se incluye en la gramática del operador `new`. Este campo permite que los nuevos objetos se inicialicen con constructores definidos por el usuario. Para obtener más información sobre cómo se realiza la inicialización, consulte [Inicializadores](#). En el ejemplo siguiente se muestra la forma de usar una expresión de inicialización con el operador `new`:

C++

```

// exre_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 };

```

```
// ...  
}
```

En este ejemplo, el objeto `CheckingAcct` se asigna usando el operador `new`, pero no se especifica ninguna inicialización predeterminada. De esta forma, se llama al constructor predeterminado para la clase `Acct()`. El objeto `SavingsAcct` se asigna de la misma manera, salvo que se inicializa explícitamente en 34,98. Dado que 34,98 es de tipo `double`, se llama al constructor que toma un argumento de ese tipo para controlar la inicialización. Finalmente, el tipo `HowMuch` que no es de clase se inicializa en 43,0.

Si un objeto es de un tipo de clase y esa clase tiene constructores (como en el ejemplo anterior), el objeto se puede inicializar mediante el operador `new` solo si se cumple una de estas condiciones:

- Los argumentos proporcionados en el inicializador concuerdan con los argumentos de un constructor.
- La clase tiene un constructor predeterminado (un constructor al que se puede llamar sin argumentos).

No se puede realizar ninguna inicialización explícita por elemento al asignar matrices mediante el operador `new`; solo se llama al constructor predeterminado, si existe. Para obtener más información, consulte [Argumentos predeterminados](#).

Si se produce un error en la asignación de memoria (`operator new` devuelve un valor de 0), no se realiza ninguna inicialización. Este comportamiento protege contra intentos de inicializar datos que no existen.

Como sucede con las llamadas a funciones, no se define el orden en que se evalúan las expresiones inicializadas. Además, no se debe dar por hecho que estas expresiones se hayan evaluado totalmente antes de que se produzca la asignación de memoria. Si se produce un error en la asignación de memoria y el operador `new` devuelve cero, es posible que algunas expresiones del inicializador no se hayan evaluado totalmente.

Duración de objetos asignados con `new`

Los objetos asignados con el operador `new` no se destruyen cuando se sale del ámbito en el que se definen. Como el operador `new` devuelve un puntero a los objetos que asigna, el programa debe definir un puntero con el ámbito adecuado para tener acceso y eliminar esos objetos. Por ejemplo:

```

// expre_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.
}

```

Una vez que el puntero `AnotherArray` sale del ámbito en el ejemplo, el objeto ya no se puede eliminar.

Cómo funciona `new`

La `new-expression` (expresión que contiene el operador `new`) hace tres cosas:

- Busca y reserva el almacenamiento para el objeto o los objetos a asignar. Cuando se completa esta fase, se asigna la cantidad correcta de almacenamiento, pero todavía no es un objeto.
- Inicializa los objetos. Una vez completada la inicialización, hay suficiente información presente para que el almacenamiento asignado sea un objeto.
- Devuelve un puntero a los objetos de un tipo de puntero derivado de `new-type-id` a `type-id`. El programa usa este puntero para tener acceso al objeto recién asignado.

El operador `new` invoca la función `operator new`. Para las matrices de cualquier tipo y para los objetos que no son de tipo `class`, `struct` o `union`, se llama a una función global, `::operator new`, para asignar almacenamiento. Los objetos de tipo de clase pueden definir su propia función de miembro estática `operator new` clase por clase.

Cuando el compilador se encuentra con el operador `new` que asigna un objeto de tipo `T`, emite una llamada a `T::operator new(sizeof(T))` o, si no hay ningún `operator new`

definido por el usuario, a `::operator new(sizeof(T))`. Así es como el operador `new` puede asignar la cantidad de memoria correcta para el objeto.

ⓘ Nota

El argumento para `operator new` es de tipo `std::size_t`. Este tipo se define en `<direct.h>`, `<malloc.h>`, `<memory.h>`, `<search.h>`, `<stddef.h>`, `<stdio.h>`, `<stdlib.h>`, `<string.h>`, and `<time.h>`.

Una opción de la gramática permite la especificación de `new-placement` (consulte la gramática para [new Operador](#)). El parámetro `new-placement` solo se puede utilizar para las implementaciones definidas por el usuario de `operator new`; permite pasar información adicional a `operator new`. Una expresión con un campo `new-placement` tal como `T *TObject = new (0x0040) T;` se convierte en `T *TObject = T::operator new(sizeof(T), 0x0040);` si la clase T tiene el miembro `operator new`; si no, se convierte en `T *TObject = ::operator new(sizeof(T), 0x0040);`.

La intención original del campo `new-placement` es permitir que los objetos dependientes de hardware se asignen a direcciones especificadas por el usuario.

ⓘ Nota

Aunque en el ejemplo anterior solo se muestra un argumento en el campo `new-placement`, no hay ninguna restricción sobre cuántos argumentos adicionales se pueden pasar a `operator new` de esta forma.

Incluso cuando `operator new` se ha definido para un tipo de clase `T`, se puede usar explícitamente el operador global `new`, como en este ejemplo:

C++

```
T *TObject = ::new TObject;
```

El operador de resolución de ámbito (`::`) fuerza el uso del operador global `new`.

Consulte también

[Expresiones con operadores unarios](#)
[Palabras clave](#)

operadores new y delete

Operador de complemento de uno: ~

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Sintaxis

C++

```
~ cast-expression
```

Comentarios

El operador de complemento de uno (~), denominado a veces operador de *complemento bit a bit*, produce un complemento de uno bit a bit de su operando. Es decir, cada bit que es 1 en el operando es 0 en el resultado. Y a la inversa: cada bit que es 0 en el operando es 1 en el resultado. El operando del operador de complemento de uno debe ser de tipo entero.

Palabra clave del operador para ~

C++ especifica `comp1` como una ortografía alternativa para ~. En C, la ortografía alternativa se proporciona como una macro en el encabezado `<iso646.h>`. En C++, la ortografía alternativa es una palabra clave; el uso de `<iso646.h>` o el equivalente de C++ `<ciso646>` está en desuso. En Microsoft C++, se requiere la opción del compilador `/permissive-` o `/Za` para poder habilitar la ortografía alternativa.

Ejemplo

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;  
}
```

En este ejemplo, el nuevo valor asignado a `y` es el complemento de uno del valor sin signo 0xFFFF, or 0x0000.

La promoción de entero se realiza en operandos enteros. El tipo al que se promueve el operando es el tipo resultante. Para obtener más información sobre la promoción integral, consulte [Conversiones estándar](#).

Consulte también

[Expresiones con operadores unarios](#)

[Operadores integrados de C++, precedencia y asociatividad](#)

[Operadores aritméticos unarios](#)

Operadores de puntero a miembro: `.*` y `->*`

`->*`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Sintaxis

pm-expression:

cast-expression

pm-expression `.*` *cast-expression*

pm-expression `->*` *cast-expression*

Comentarios

Los operadores de puntero a miembro `.*` y `->*` devuelven el valor de un miembro de clase concreto para el objeto especificado en el lado izquierdo de la expresión. El lado derecho debe especificar un miembro de la clase. En el siguiente ejemplo se muestra cómo usar estos operadores.

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;
}

```

Output

Output

```

m_func1
m_func1
1
2

```

En el ejemplo anterior, se utiliza un puntero a un miembro, `pmfn`, para invocar la función miembro `m_func1`. Se usa otro puntero a miembro, `pmd`, para tener acceso al miembro `m_num`.

El operador binario `.*` combina su primer operando, que debe ser un objeto de tipo de clase, con su segundo operando, que debe ser un tipo de puntero a miembro.

El operador binario `->*` combina su primer operando, que debe ser un puntero a un objeto de tipo de clase, con su segundo operando, que debe ser un tipo de puntero a miembro.

En una expresión que contenga el operador `.*`, el primer operando debe ser del tipo de clase y ser accesible para el puntero al miembro especificado en el segundo operando, o bien de un tipo accesible derivado inequívocamente de y accesible para esa clase.

En una expresión que contenga el operador `->*`, el primer operando debe ser de tipo "puntero al tipo de clase" del tipo especificado en el segundo operando, o bien debe ser de un tipo derivado inequívocamente de esa clase.

Ejemplo

Considere las clases y el fragmento de programa siguientes:

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.
}
```

El resultado de los operadores de puntero a miembro `.*` o `->*` es un objeto o una función del tipo especificado en la declaración del puntero a miembro. Por lo tanto, en el ejemplo anterior, el resultado de la expresión `ADerived.*pmfnFunc1()` es un puntero a una función que devuelve `void`. El resultado es un valor L si el segundo operando es un valor L.

ⓘ Nota

Si el resultado de uno de los operadores de puntero a miembro es una función, el resultado se puede utilizar como operando al operador de llamada a función.

Consulte también

[Operadores integrados de C++, precedencia y asociatividad](#)

Operadores de incremento y decremento postfijos: ++ y --

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Sintaxis

```
postfix-expression ++
postfix-expression --
```

Comentarios

C++ proporciona operadores de incremento y decremento tanto de prefijo como de postfijo. En esta sección, se describen solo los de postfijo. (Para más información, consulte [Operadores de incremento y decremento prefijos](#)). La diferencia entre los dos es que, en la notación postfija, el operador aparece después de *postfix-expression*, mientras que, en la notación prefija, el operador aparece antes de *expression*. En el ejemplo siguiente, se muestra un operador de decremento postfijo:

```
C++
```

```
i++;
```

La aplicación del operador de incremento postfijo (++) tiene como efecto que el valor del operando se incrementa en una unidad del tipo adecuado. De manera similar, la aplicación del operador de decremento postfijo (--) tiene como efecto que el valor del operando se reduce en una unidad del tipo adecuado.

Es importante tener en cuenta que una expresión de incremento o decremento postfijo se evalúa como el valor de la expresión *antes de* aplicar el operador correspondiente. La operación de incremento o decremento se produce *después* de que se evalúa el operando. El problema surge solo cuando la operación de incremento o decremento de postfijo se da en el contexto de una expresión mayor.

Cuando se aplica un operador de postfijo a un argumento de función, no es seguro que el valor del argumento aumente o disminuya antes de que se pase a la función. Para obtener más información, vea la sección 1.9.17 del estándar C++.

Al aplicar el operador de incremento postfijo a un puntero en una matriz de objetos de tipo `long`, en realidad se agregan cuatro a la representación interna del puntero. Este comportamiento hace que el puntero, que antes hacía referencia al elemento n de la matriz, haga referencia al elemento $(n+1)$.

Los operandos de los operadores de decremento e incremento postfijo deben ser valores L modificables (no `const`) de tipo aritmético o puntero. El tipo del resultado es el mismo que el de *postfix-expression*, pero ya no es un valor L.

Visual Studio 2017, versión 15.3 y posteriores (disponible en modo `/std:c++17` y versiones posteriores): es posible que un operando de un operador de incremento o decremento postfijo no sea de tipo `bool`.

El código siguiente muestra el operador de incremento de postfijo:

C++

```
// expre_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;
}
```

No se admiten las operaciones de incremento y decremento de postfijo en los tipos enumerados:

C++

```
enum Compass { North, South, East, West };
Compass myCompass;
for( myCompass = North; myCompass != West; myCompass++ ) // Error
```

Consulte también

[Expresiones postfijas](#)

[Operadores integrados de C++, precedencia y asociatividad](#)

[Operadores de incremento y decremento postfijos de C](#)

Operadores de incremento y decremento prefijos: ++ y --

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Sintaxis

```
++ unary-expression  
-- unary-expression
```

Comentarios

El operador de incremento de prefijo (++) suma uno a su operando; este valor incrementado es el resultado de la expresión. El operando debe ser un valor l distinto del tipo `const`. El resultado es un valor l del mismo tipo que el operando.

El operador de decremento de prefijo (--) es similar al operador de incremento de prefijo, solo que el operando se reduce en uno y el resultado es este valor disminuido.

Visual Studio 2017, versión 15.3 y posteriores (disponible en `/std:c++17` modo y versiones posteriores): es posible que el operando de un operador de incremento o decremento no sea de tipo `bool`.

Los operadores de incremento y decremento de prefijo y postfijo afectan a sus operandos. La principal diferencia entre ellos es el orden en que se realiza el incremento o el decremento al evaluar una expresión. (Para más información, vea [Operadores de incremento y decremento de postfijos](#).) En la forma de prefijo, el incremento o decremento tiene lugar antes de que el valor se use en la evaluación de la expresión, por lo que el valor de la expresión es diferente del valor del operando. En la forma de postfijo, el incremento o decremento tiene lugar después de que el valor se use en la evaluación de la expresión, por lo que el valor de la expresión es el mismo que el valor del operando. Por ejemplo, el siguiente programa imprime “`++i = 6`”:

C++

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

```
using namespace std;

int main() {
    int i = 5;
    cout << "++i = " << ++i << endl;
}
```

Un operando de tipo entero o flotante aumenta o disminuye en el valor entero 1. El tipo del resultado es el mismo que el tipo del operando. Un operando de tipo de puntero aumenta o disminuye en el tamaño del objeto al que apunta. Un puntero incrementado apunta al siguiente objeto; un puntero disminuido apunta al objeto anterior.

Como los operadores de incremento y decremento tienen efectos secundarios, el uso de expresiones con operadores de incremento y decremento en una [macro de preprocesador](#) puede producir resultados no deseados. Considere este ejemplo:

C++

```
// expe_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 );
}
```

La macro se expande en:

C++

```
k = ((++i)<(j))?(j):(++i);
```

Si `i` es mayor o igual que `j` o es menor que `j` en 1, aumentará dos veces.

ⓘ Nota

Las funciones insertadas de C++ son preferibles a las macros en muchos casos porque eliminan los efectos secundarios como los que se describen aquí, y permiten que el lenguaje realice una comprobación de tipos más completa.

Vea también

Expresiones con operadores unarios

Operadores integrados de C++, precedencia y asociatividad

Operadores de incremento y decremento prefijos

Operadores relacionales: <, >, <= y >=

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Sintaxis

```
expression < expression
expression > expression
expression <= expression
expression >= expression
```

Comentarios

Los operadores relacionales binarios determinan las relaciones siguientes:

- Menor que (<)
- Mayor que (>)
- Menor o igual que (<=)
- Mayor o igual que (>=)

Los operadores relacionales tienen asociatividad de izquierda a derecha. Ambos operandos de los operadores relacionales deben ser de tipo aritmética o puntero. Producen valores de tipo `bool`. El valor devuelto es `false` (0) si la relación de la expresión es falsa; si no, el valor devuelto es `true` (1).

Ejemplo

C++

```
// expe_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;  
}
```

Las expresiones del ejemplo anterior se deben incluir entre paréntesis porque el operador de inserción de la secuencia (`<<`) tiene mayor prioridad que los operadores relacionales. Por consiguiente, la primera expresión sin paréntesis se evaluaría como:

C++

```
(cout << "The true expression 3 > 2 yields: " << 3) < (2 << "\n");
```

Las conversiones aritméticas habituales tratadas en [Conversiones estándar](#) se aplican a los operandos de tipos aritméticos.

Comparar punteros

Cuando se comparan dos punteros a objetos del mismo tipo, el resultado está determinado por la ubicación de los objetos a los que se señala en el espacio de direcciones del programa. Los punteros también se pueden comparar con una expresión de constante que se evalúa como 0 o con un puntero de tipo `void *`. Si una comparación de puntero se realiza frente a un puntero de tipo `void *`, el otro puntero se convierte implícitamente al tipo `void *`. A continuación, se realiza la comparación.

Dos punteros de tipos diferentes no pueden compararse a menos que:

- Un tipo sea un tipo de clase derivado del otro tipo.
- Al menos uno de los punteros se convierta explícitamente (conversión) al tipo `void *`. (El otro puntero se convierte implícitamente al tipo `void *` para la conversión).

Se garantiza que dos punteros del mismo tipo que señalan al mismo objeto realizan una comparación igual. Si se comparan dos punteros a miembros no estáticos de un objeto, se aplican las reglas siguientes:

- Si el tipo de clase no es una `union` y si los dos miembros no se separan mediante un *access-specifier* (especificador de acceso), como `public`, `protected` o `private`, el puntero al miembro declarado en último lugar realizará una comparación mayor que el puntero al miembro declarado anteriormente.
- Si los dos miembros están separados por un *access-specifier*, los resultados son indefinidos.

- Si el tipo de clase es una `union`, los punteros a miembros de datos diferentes en esa `union` realizan una comparación igual.

Si dos punteros señalan a elementos de la misma matriz o al elemento uno situado más allá del final de la matriz, el puntero al objeto con el subíndice más alto realiza una comparación superior. Se garantiza que la comparación de punteros es válida solo cuando los punteros hacen referencia a objetos de la misma matriz o a la ubicación uno superado el final de la matriz.

Consulte también

[Expresiones con operadores binarios](#)

[Operadores integrados de C++, precedencia y asociatividad](#)

[Operadores relacionales y de igualdad de C](#)

Operador de resolución de ámbito ::

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

El operador de resolución de ámbito :: se usa para identificar los identificadores usados en distintos ámbitos y eliminar la ambigüedad entre ellos. Para obtener más información sobre el ámbito, consulte [Ámbito](#).

Sintaxis

qualified-id:

nested-name-specifier template opt *unqualified-id*

nested-name-specifier:

::

type-name ::

namespace-name ::

decltype-specifier ::

nested-name-specifier identifier ::

nested-name-specifier template opt *simple-template-id* ::

unqualified-id:

identifier

operator-function-id

conversion-function-id

literal-operator-id

~ *type-name*

~ *decltype-specifier*

template-id

Comentarios

El *identifier* puede ser una variable, una función o un valor de enumeración.

Uso de :: para clases y espacios de nombres

En el ejemplo siguiente se muestra cómo se usa el operador de resolución con clases y espacios de nombres:

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;
}
```

Un operador de resolución de ámbito sin un calificador de ámbito hace referencia al espacio de nombres global.

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;
}
```

Puede usar el operador de resolución de ámbito para identificar un miembro de o **namespace** para identificar un espacio de nombres que designe el espacio de nombres del miembro en una **using** directiva . En el ejemplo siguiente, se puede usar **NamespaceC**

para calificar `ClassB` a pesar de que `ClassB` se declaró en el espacio de nombres `NamespaceB`, ya que se mencionó `NamespaceB` en `NamespaceC` mediante una directiva `using`.

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;  
}
```

Se pueden usar cadenas de operadores de resolución de ámbito. En el ejemplo siguiente, `NamespaceD::NamespaceD1` identifica el espacio de nombres anidado `NamespaceD1` y `NamespaceE::ClassE::ClassE1` identifica la clase anidada `ClassE1`.

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 ;  
}
```

Uso de :: para miembros estáticos

Debe usar el operador de resolución de ámbito para llamar a miembros estáticos de clases.

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();  
}
```

Uso de :: para enumeraciones de ámbito

El operador de resolución de ámbito se usa también con los valores de una enumeración de ámbito en [declaraciones de enumeración](#), como en el ejemplo siguiente:

C++

```
enum class EnumA{  
    First,  
    Second,  
    Third  
};  
  
int main() {  
    EnumA enum_value = EnumA::First;  
}
```

Consulte también

Operadores integrados de C++, precedencia y asociatividad

Espacios de nombres

sizeof (Operador)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Genera el tamaño de su operando con respecto al tamaño de tipo `char`.

ⓘ Nota

Para obtener información sobre el operador `sizeof ...`, consulte [Puntos suspensivos y plantillas variádicas](#).

Sintaxis

```
sizeof unary-expression  
sizeof ( type-name )
```

Comentarios

El resultado del operador `sizeof` es de tipo `size_t`, un tipo entero definido en el archivo de inclusión `<stddef.h>`. Este operador permite no tener que especificar tamaños de datos dependientes del equipo en los programas.

El operando para `sizeof` puede ser uno de los siguientes:

- Nombre de tipo. Para utilizar `sizeof` con un nombre de tipo, el nombre debe ir entre paréntesis.
- Expresión. Cuando se utiliza con una expresión, se puede especificar `sizeof` con o sin paréntesis. La expresión no se evalúa.

Cuando el operador `sizeof` se aplica a un objeto de tipo `char`, genera 1. Cuando el operador `sizeof` se aplica a una matriz, genera el número total de bytes de esa matriz, no el tamaño del puntero representado por el identificador de matriz. Para obtener el tamaño del puntero representado por el identificador de matriz, páselo como parámetro a una función que utilice `sizeof`. Por ejemplo:

Ejemplo

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;
}
```

Salida de ejemplo

Output

```
The size of a char is: 1
The length of Hello, world! is: 14
The size of the pointer is 4
```

Cuando el operador `sizeof` se aplica a un tipo `class`, `struct` o `union`, el resultado es el número de bytes de un objeto de ese tipo, además del relleno que se agregue para alinear los miembros en los límites de palabra. El resultado no corresponde necesariamente al tamaño que se calcula agregando los requisitos de almacenamiento de los miembros individuales. La opción `/Zp` del compilador y la directiva pragma `pack` afectan a los límites de alineación de los miembros.

El operador `sizeof` nunca genera 0, incluso para una clase vacía.

El operador `sizeof` no se puede utilizar con los operandos siguientes:

- Funciones. (Sin embargo, `sizeof` se puede aplicar a punteros a funciones).
- Campos de bits.
- Clases no definidas.

- El tipo `void`.
- Matrices asignadas dinámicamente.
- Matrices externas.
- Tipos incompletos.
- Nombres entre paréntesis de tipos incompletos.

Cuando el operador `sizeof` se aplica a una referencia, el resultado es el mismo que si se hubiera aplicado `sizeof` al propio objeto.

Si una matriz sin tamaño es el último elemento de una estructura, el operador `sizeof` devuelve el tamaño de la estructura sin la matriz.

El operador `sizeof` suele utilizarse para calcular el número de elementos de una matriz mediante una expresión con este formato:

C++

```
sizeof array / sizeof array[0]
```

Vea también

[Expresiones con operadores unarios](#)

[Palabras clave](#)

Operador de subíndice []

Artículo • 26/09/2022 • Tiempo de lectura: 3 minutos

Sintaxis

```
postfix-expression [ expression ]
```

Comentarios

Una expresión de postfijo (que también puede ser una expresión primaria) seguida del operador de subíndice, [], especifica la indexación de matrices.

Para obtener información sobre las matrices administradas en C++/CLI, consulte [Matrices](#).

Normalmente, el valor representado por *postfix-expression* es un valor de puntero, como un identificador de matriz, y *expression* es un valor entero (incluidos los tipos enumerados). Sin embargo, todo lo que se necesita desde el punto de vista sintáctico es que una de las expresiones sea de tipo puntero y que la otra sea de tipo entero. Así pues, el valor integral podría estar en la posición de *postfix-expression* y el valor de puntero podría estar en los corchetes de la posición de *expression* o subíndice. Observe el fragmento de código siguiente:

C++

```
int nArray[5] = { 0, 1, 2, 3, 4 };
cout << nArray[2] << endl;           // prints "2"
cout << 2[nArray] << endl;          // prints "2"
```

En el ejemplo anterior, la expresión `nArray[2]` es idéntica a `2[nArray]`. La razón es que el resultado de una expresión de subíndice `e1[e2]` viene determinado por:

`*((e2) + (e1))`

La dirección producida por la expresión no es *e2* bytes desde la dirección *e1*. En su lugar, la dirección se escala para producir el siguiente objeto en la matriz *e2*. Por ejemplo:

C++

```
double aDb[2];
```

Las direcciones de `aDb[0]` y `aDb[1]` están alejadas 8 bytes: el tamaño de un objeto de tipo `double`. Este escalado según el tipo de objeto se realiza automáticamente en el lenguaje C++ y se define en [Operadores de suma](#) donde se describe la suma y resta de operandos de tipo puntero.

Una expresión de subíndice también puede tener varios subíndices, como se indica a continuación:

expression1[expression2] [expression3] ...

Las expresiones de subíndice se asocian de izquierda a derecha. La expresión de subíndice del extremo izquierdo, `expression1[expression2]`, se evalúa primero. La dirección resultante de agregar `expression1` y `expression2` forma una expresión de puntero; entonces, se agrega `expression3` a esta expresión de puntero para formar una nueva expresión de puntero, y así sucesivamente, hasta que se haya agregado la última expresión de subíndice. El operador de direccionamiento indirecto (*) se aplica después de que evalúe la última expresión de subíndice, a menos que el valor del puntero final apunte a un tipo de matriz.

Las expresiones con varios subíndices hacen referencia a elementos de matrices multidimensionales. Una matriz multidimensional es una matriz cuyos elementos son matrices. Por ejemplo, el primer elemento de una matriz tridimensional es una matriz con dos dimensiones. En el ejemplo siguiente se declara y se inicializa una matriz bidimensional de caracteres simple:

C++

```
// expre_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;
}
```

Subíndices positivos y negativos

El primer elemento de una matriz es el elemento 0. El intervalo de una matriz de C++ es de `array[0]` a `array[size - 1]`. Sin embargo, C++ admite subíndices positivos y negativos. Los subíndices negativos deben situarse dentro de los límites de la matriz, ya que de lo contrario los resultados son impredecibles. En el código siguiente se muestran subíndices de matriz positivos y negativos:

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
}
```

El subíndice negativo de la última línea puede generar un error en tiempo de ejecución porque señala a una dirección 256 `int` posiciones más abajo en la memoria que el origen de la matriz. El puntero `midArray` se inicializa en el centro de `intArray`, por lo que en él (pero peligroso) se pueden usar índices de matriz positivos y negativos. Los errores de subíndice de matriz no generan errores en tiempo de compilación, pero producen resultados imprevisibles.

El operador de subíndice es commutativo. Por consiguiente, está garantizado que las expresiones `array[index]` e `index[array]` son equivalentes siempre que el operador subíndice no esté sobrecargado (consulte [Operadores sobrecargados](#)). La primera forma es la práctica más común de codificación, pero cualquiera de ellas funciona.

Consulte también

Expresiones postfijas

Operadores integrados de C++, precedencia y asociatividad

Matrices

Matrices unidimensionales

Matrices multidimensionales

typeid (Operador)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Sintaxis

```
typeid(type-id)
typeid(expression)
```

Comentarios

El operador `typeid` permite que se determine el tipo de un objeto en tiempo de ejecución.

Resultado de `typeid` es un `const type_info&`. El valor es una referencia a un objeto `type_info` que representa el *type-id* o el tipo de *expression*, dependiendo de la forma en que se use `typeid`. Para más información, consulte [Clase type_info](#).

El operador `typeid` no funciona con tipos administrados (declaradores abstractos o instancias). Para obtener información sobre cómo obtener el [Type](#) de un tipo especificado, consulte [typeid](#).

El operador `typeid` realiza una comprobación en tiempo de ejecución cuando se aplica a un valor de un tipo de clase polimórfico, en que el tipo verdadero del objeto no se puede determinar mediante la información estática proporcionada. Esos casos son:

- Una referencia a una clase
- Un puntero, desreferenciado con `*`
- Un puntero con subíndice (`[]`). (No es seguro usar un subíndice con un puntero a un tipo polimórfico).

Si *expression* señala a un tipo de clase base, aunque el objeto sea realmente de un tipo derivado de esa clase base, el resultado es una referencia de `type_info` para la clase derivada. El elemento *expression* debe señalar a un tipo polimórfico (una clase con funciones virtuales). De lo contrario, el resultado es el `type_info` para la clase estática a que se hace referencia en la *expression*. Además, el puntero se debe desreferenciar para

que el objeto usado sea el que señala. Si no se desreferencia el puntero, el resultado será el `type_info` para el puntero, no el objeto al que señala. Por ejemplo:

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;
}
```

Si la *expression* desreferencia un puntero y el valor de ese puntero es cero, `typeid` produce una excepción `bad_typeid`. Si el puntero no apunta a un objeto válido, se produce una excepción `_non_rtti_object`. Indica un intento de analizar el RTTI que desencadenó un error porque el objeto no es válido de alguna manera. (Por ejemplo, es un puntero incorrecto o el código no se compiló con `/GR`).

Si la *expression* no es un puntero ni una referencia a una clase base del objeto, el resultado es una referencia de `type_info` que representa el tipo estático de la *expresión*. El elemento *static type* de una expresión se refiere al tipo de una expresión cuando se conoce en el tiempo de compilación. La semántica de ejecución se omite cuando se evalúa el tipo estático de una expresión. Además, las referencias se omiten cuando es posible al determinar el tipo estático de una expresión:

C++

```
// expre_typeid_Operator_2.cpp
#include <typeinfo>

int main()
{
```

```
    typeid(int) == typeid(int&); // evaluates to true
}
```

`typeid` también se puede utilizar en las plantillas para determinar el tipo de un parámetro de plantilla:

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 );
}
```

Vea también

[Información de tipos en tiempo de ejecución](#)

[Palabras clave](#)

Operadores unarios más y de negación: + y -

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Sintaxis

```
+ cast-expression  
- cast-expression
```

+ (operador)

El resultado del operador unario más (+) es el valor de su operando. El operando del operador unario más debe ser de tipo aritmético.

La promoción de entero se realiza en operandos enteros. El tipo resultante es el tipo al que se promueve el operando. Así, la expresión `+ch`, donde `ch` es de tipo `char`, produce el tipo `int`; el valor está sin modificar. Consulte [Conversiones Estándar](#) para obtener más información sobre cómo se realiza la promoción.

- (operador)

El operador de negación unario (-) genera el negativo de su operando. El operando del operador de negación unario debe ser un tipo aritmético.

La promoción de entero se realiza en operandos enteros y el tipo resultante es el tipo al que se promueve el operando. Consulte [Conversiones Estándar](#) para obtener más información sobre cómo se ejecuta la promoción.

Específicos de Microsoft

La negación unaria de cantidades sin signo se realiza restando el valor del operando de 2^n , donde n es el número de bits de un objeto del tipo sin signo especificado.

FIN de Específicos de Microsoft

Vea también

Expresiones con operadores unarios

Operadores integrados de C++, precedencia y asociatividad

Expresiones (C++)

Artículo • 26/09/2022 • Tiempo de lectura: 2 minutos

En esta sección se describen las expresiones de C++. Las expresiones son secuencias de operadores y operandos que se utilizan para uno o más de estos propósitos:

- Calcular un valor a partir de los operandos.
- Designar objetos o funciones.
- Generar “efectos secundarios”. (Los efectos secundarios son cualquier acción distinta de la evaluación de la expresión; por ejemplo, modificando el valor de un objeto).

En C++, los operadores se pueden sobrecargar y el usuario puede definir sus significados. Sin embargo, la prioridad y el número de operandos que aceptan no pueden modificarse. En esta sección se describe la sintaxis y la semántica de los operadores tal como se proporcionan con el lenguaje, no sobrecargados. Además de los [tipos de expresiones](#) y la [semántica de las expresiones](#), se tratan los siguientes temas:

- [Expresiones primarias](#)
- [Operador de resolución de ámbito](#)
- [Expresiones de postfijo](#)
- [Expresiones con operadores unarios](#)
- [Expresiones con operadores binarios](#)
- [Operador condicional](#)
- [Expresiones constantes](#)
- [Operadores de conversión](#)
- [Información de tipos en tiempo de ejecución](#)

Temas sobre operadores en otras secciones:

- [Operadores integrados de C++, precedencia y asociatividad](#)
- [Operadores sobrecargados](#)
- [typeid \(C++/CLI\)](#)

 **Nota**

Los operadores para los tipos integrados no se pueden sobrecargar; su comportamiento está predefinido.

Vea también

[Referencia del lenguaje C++](#)

Tipos de expresiones

Artículo • 26/09/2022 • Tiempo de lectura: 2 minutos

Las expresiones de C++ se dividen en varias categorías:

- **Expresiones primarias.** Son los bloques de creación con los que se forman las demás expresiones.
- **Expresiones de postfijo.** Son expresiones primarias seguidas de un operador (por ejemplo, el subíndice de la matriz o el operador de incremento de postfijo).
- **Expresiones formadas con operadores unarios.** Los operadores unarios actúan solo sobre un operando en una expresión.
- **Expresiones formadas con operadores binarios.** Los operadores binarios actúan sobre dos operandos de una expresión.
- **Expresiones con el operador condicional.** El operador condicional es un operador ternario (el único del lenguaje C++) que utiliza tres operandos.
- **Expresiones de constante.** Las expresiones de constante se forman completamente con datos constantes.
- **Expresiones con conversiones de tipos explícitas.** En las expresiones se pueden usar conversiones de tipos explícitas.
- **Expresiones con operadores de puntero a miembro.**
- **Conversión.** En las expresiones se pueden usar conversiones con seguridad de tipos.
- **Información de tipos en tiempo de ejecución.** Determine el tipo de un objeto durante la ejecución del programa.

Consulte también

[Expresiones](#)

Expresiones primarias

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Las expresiones primarias son los bloques de creación de expresiones más complejas. Pueden ser literales, nombres, y nombres calificados por el operador de resolución de ámbito (::). Una expresión primaria puede tener cualquiera de las formas siguientes:

```
primary-expression
  literal
  this
  name
  :: name ( expression )
```

Un `literal` es una expresión primaria constante. Su tipo depende de la forma de su especificación. Consulte [Literales](#) para obtener información completa sobre la especificación de literales.

La palabra clave `this` es un puntero a un objeto de clase. Está disponible dentro de funciones miembro no estáticas. Apunta a la instancia de la clase para la que se invocó la función. La palabra clave `this` no se puede utilizar fuera del cuerpo de una función miembro de clase.

El tipo del puntero `this` es `type * const` (donde `type` es el nombre de clase) en funciones que no modifican específicamente el puntero `this`. En el ejemplo siguiente se muestran declaraciones de funciones miembro y los tipos de `this`:

C++

```
// expe_Primary_Expressions.cpp
// compile with: /LD
class Example
{
public:
    void Func();           // * const this
    void Func() const;    // const * const this
    void Func() volatile; // volatile * const this
};
```

Para obtener más información sobre cómo modificar el tipo del puntero `this`, consulte [this puntero](#).

El operador de resolución de ámbito (::) seguido de un nombre es una expresión primaria. Dichos nombres deben ser nombres en el ámbito global, no nombres de

miembro. El tipo de la expresión está determinado por la declaración del nombre. Es un valor L (es decir, puede aparecer en el lado izquierdo de una expresión de asignación) si el nombre de declaración es un valor L. El operador de resolución de ámbito permite que se haga referencia a un nombre global, incluso si ese nombre está oculto en el ámbito actual. Consulte [Ámbito](#) para obtener un ejemplo de cómo se utiliza el operador de resolución de ámbito.

Una expresión entre paréntesis es una expresión primaria. Su tipo y valor son idénticos al tipo y el valor de la expresión sin parentesco. Es un valor L si la expresión no incluida entre paréntesis es un valor L.

Entre los ejemplos de expresiones primarias se incluyen:

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
```

Estos ejemplos se consideran *nombres* todos ellos, y por lo tanto son expresiones primarias, en diversas formas:

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
```

Consulte también

[Tipos de expresiones](#)

Puntos suspensivos y plantillas variádicas

Artículo • 03/03/2023 • Tiempo de lectura: 4 minutos

En este artículo se muestra cómo utilizar los puntos suspensivos (...) con plantillas variádicas de C++. Los puntos suspensivos han tenido muchos usos en C y C++. Entre ellos se incluyen listas de argumentos de variable para funciones. La función `printf()` de la biblioteca en tiempo de ejecución de C es uno de los ejemplos más conocidos.

Una *plantilla variádica* es una plantilla de clase o de función que admite un número arbitrario de argumentos. Este mecanismo es especialmente útil para los desarrolladores de bibliotecas de C++: puede aplicarlo a plantillas de clase y plantillas de función, y, por tanto, proporcionar una amplia gama de funcionalidades y flexibilidad no triviales y seguras para tipos.

Sintaxis

Las plantillas variádicas utilizan los puntos suspensivos de dos maneras. A la izquierda del nombre de parámetro, significa un *paquete de parámetros*, y a la derecha del nombre de parámetro, expande los paquetes de parámetros en nombres diferentes.

Este es un ejemplo básico de sintaxis de definición de *plantilla de clase variádica*:

C++

```
template<typename... Arguments> class classname;
```

Para los paquetes de parámetros y las expansiones, puede agregar espacios en blanco alrededor de los puntos suspensivos, según sus preferencias, como se muestra en este ejemplo:

C++

```
template<typename ...Arguments> class classname;
```

O este ejemplo:

C++

```
template<typename ... Arguments> class classname;
```

En este artículo se usa la convención que se muestra en el primer ejemplo (los puntos suspensivos se adjuntan a `typename`).

En los ejemplos anteriores, `Arguments` es un paquete de parámetros. La clase `classname` puede aceptar un número variable de argumentos, como en estos ejemplos:

C++

```
template<typename... Arguments> class vtclass;

vtclass<> vtinstance1;
vtclass<int> vtinstance2;
vtclass<float, bool> vtinstance3;
vtclass<long, std::vector<int>, std::string> vtinstance4;
```

Mediante el uso de una definición de plantilla de clase variádica, también puede requerir al menos un parámetro:

C++

```
template <typename First, typename... Rest> class classname;
```

Este es un ejemplo básico de sintaxis de *plantilla de función variádica*:

C++

```
template <typename... Arguments> returntype functionname(Arguments... args);
```

A continuación, el `Arguments` paquete de parámetros se expande para su uso, como se muestra en la sección siguiente.

Otras formas de sintaxis de plantilla de función variádica son posibles, incluidos, entre otros, estos ejemplos:

C++

```
template <typename... Arguments> returntype functionname(Arguments&... args);
template <typename... Arguments> returntype functionname(Arguments&&... args);
template <typename... Arguments> returntype functionname(Arguments*... args);
```

También se permiten especificadores como `const`:

C++

```
template <typename... Arguments> returntype functionname(const Arguments&... args);
```

Como ocurre con las definiciones de clase de plantilla variádica, se pueden crear funciones que requieran al menos un parámetro:

C++

```
template <typename First, typename... Rest> returntype functionname(const First& first, const Rest&... args);
```

Las plantillas variádicas utilizan el operador `sizeof...()` (relacionado con el anterior operador `sizeof()`):

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...);
}
```

Más información sobre la posición de los puntos suspensivos

Anteriormente, en este artículo se describió la colocación de puntos suspensivos que define paquetes de parámetros y expansiones en este formato: "a la izquierda del nombre del parámetro, significa un paquete de parámetros y, a la derecha del nombre del parámetro, expande los paquetes de parámetros en nombres independientes". Aunque técnicamente es cierto, puede resultar confuso en la traducción al código. Tenga en cuenta lo siguiente:

- En una lista de parámetros de plantilla (`template <parameter-list>`), `typename...` introduce un paquete de parámetros de plantilla.
- En una cláusula de declaración de parámetros (`func(parameter-list)`), los puntos suspensivos "de nivel superior" introducen un paquete de parámetros de función y

la posición de los puntos suspensivos es importante:

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);
```

- Cuando los puntos suspensivos aparecen inmediatamente después de un nombre de parámetro, tiene una expansión del paquete de parámetros.

Ejemplo

Una buena manera de ilustrar el mecanismo de plantilla de función variádica es usarlo en una reescritura de algunas de las funciones de `printf`:

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);  
}
```

Resultados

Output

```
1
10, 20
100, 200, 300
first, 2, third, 3.14159
```

ⓘ Nota

La mayoría de las implementaciones que incorporan plantillas de función variádicas usan recursividad de alguna forma, pero es ligeramente diferente de la recursividad tradicional. La recursión tradicional implica que una función se llame a sí misma al usar la misma firma. (Puede estar sobrecargado o ser una plantilla, pero se elige la misma firma cada vez). La recursividad variádica consiste en llamar a una plantilla de función variádica mediante el uso de números diferentes de argumentos (casi siempre decrecientes) y así imprimir una firma diferente cada vez. Sigue siendo necesario un "caso base", pero la naturaleza de la recursividad es diferente.

Expresiones postfijas

Artículo • 03/03/2023 • Tiempo de lectura: 7 minutos

Las expresiones de postfijo constan de expresiones primarias o expresiones en las que los operadores de postfijo siguen una expresión primaria. Los operadores de postfijo se enumeran en la tabla siguiente.

Operadores de postfijo

| Nombre de operador | Notación de operador |
|---|--------------------------|
| Operador de subíndice | [] |
| Operador de llamada de función | () |
| Operador de conversión explícita de tipos | <i>Nombre de tipo()</i> |
| Operador de acceso a miembros | . o -> |
| Operador de incremento de postfijo | ++ |
| Operador de decremento de postfijo | -- |

La sintaxis siguiente describe expresiones de postfijo posibles:

```
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 )
```

La *expresión de postfijo* (postfix-expression) que figura arriba puede ser una [expresión primaria](#) u otra expresión de postfijo. Las expresiones de postfijo se agrupan de izquierda a derecha, por lo que las expresiones se pueden encadenar unas a otras del modo siguiente:

C++

```
func(1)->GetValue()++
```

En la expresión anterior, `func` es una expresión primaria, `func(1)` es una expresión de postfijo de función, `func(1)->GetValue` es una expresión de postfijo que especifica un

miembro de la clase, `func(1)->GetValue()` es otra expresión de postfijo de función y la expresión completa es una expresión de postfijo que incrementa el valor devuelto de `GetValue`. El significado de la expresión completa es que la función que llama pasa 1 como argumento y obtiene un puntero a una clase como valor devuelto. La llamada a `GetValue()` en esa clase incrementa después el valor devuelto.

Las expresiones enumeradas anteriormente son expresiones de asignación, lo que significa que el resultado de estas expresiones debe ser un valor R.

La forma de expresión de postfijo

C++

```
simple-type-name ( expression-list )
```

indica la invocación del constructor. Si el nombre de tipo simple es un tipo fundamental, la lista de expresiones debe ser una expresión única y esta expresión indica una conversión del valor de la expresión al tipo fundamental. Este tipo de expresión de conversión imita un constructor. Dado que esta forma permite la construcción de tipos y clases fundamentales utilizando la misma sintaxis, es especialmente útil cuando se definen clases de plantilla.

cast-keyword puede ser `dynamic_cast`, `static_cast` o `reinterpret_cast`. Se puede encontrar más información en [dynamic_cast](#), [static_cast](#) y [reinterpret_cast](#).

El operador `typeid` se considera una expresión de postfijo. Consulte [operador typeid](#).

Argumentos formales y reales

Los programas de llamada pasan la información a las funciones llamadas en "argumentos reales". Las funciones llamadas acceden a la información mediante los "argumentos formales" correspondientes.

Cuando se llama a una función, se realizan las tareas siguientes:

- Se evalúan todos los argumentos reales (los proporcionados por el llamador). No hay ningún orden implícito en el que se evalúan estos argumentos, pero se evalúan todos los argumentos y se completan todos los efectos secundarios antes de la entrada a la función.
- Cada argumento formal se inicializa con su argumento real correspondiente de la lista de expresiones. (Un argumento formal es un argumento que se declara en el encabezado de función y se usa en el cuerpo de una función). Las conversiones se

realizan como si se tratara de una inicialización, tanto las conversiones estándar como las definidas por el usuario se realizan al convertir un argumento real en el tipo correcto. Inicialización realizada se muestra de forma conceptual en el código siguiente:

```
C++  
  
void Func( int i ); // Function prototype  
...  
Func( 7 );          // Execute function call
```

Las inicializaciones conceptuales anteriores a la llamada son:

```
C++  
  
int Temp_i = 7;  
Func( Temp_i );
```

Observe que la inicialización se realiza como si se utilizara la sintaxis de signo igual en lugar de la sintaxis de paréntesis. Se realiza una copia de `i` antes de pasar el valor a la función. (Para obtener más información, consulte [Inicializadores y Conversiones](#)).

Por lo tanto, si las llamadas de prototipo de función (declaración) para un argumento de tipo `long` y si el programa que llama proporciona un argumento real de tipo `int`, el argumento real se promueve mediante una conversión de tipo standard a tipo `long` (consulte [Conversiones estándar](#)).

Es un error proporcionar un argumento real para el que no hay ninguna conversión estándar o definida por el usuario al tipo del argumento formal.

Para los argumentos reales de tipo de clase, el argumento formal se inicializa llamando al constructor de la clase. (Consulte [Constructores](#) para obtener más información sobre estas funciones miembro de clase especiales).

- Se ejecuta la llamada de función.

El fragmento de programa siguiente muestra una llamada de función:

```
C++  
  
// expte_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 )
{
}

```

Cuando se llama a `func` desde `main`, el parámetro formal `param1` se inicializa con el valor de `i` (`i` se convierte al tipo `long` que corresponde al tipo correcto mediante una conversión estándar) y el parámetro formal `param2` se inicializa con el valor de `j` (`j` se convierte al tipo `double` mediante una conversión estándar).

Tratamiento de tipos de argumento

Los argumentos formales declarados como tipos `const` no se pueden cambiar dentro del cuerpo de una función. Las funciones pueden cambiar cualquier argumento que no sea de tipo `const`. Sin embargo, el cambio es local para la función y no afecta al valor del argumento real a menos que el argumento real sea una referencia a un objeto que no sea de tipo `const`.

Las funciones siguientes muestran algunos de estos conceptos:

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;
}

```

Puntos suspensivos y argumentos predeterminados

Las funciones se pueden declarar para aceptar menos argumentos que los especificados en la definición de función, mediante uno de estos dos métodos: puntos suspensivos (...) o argumentos predeterminados.

Los puntos suspensivos denotan que los argumentos pueden ser necesarios pero no se especifica ni el número ni los tipos en la declaración. Normalmente se considera una mala práctica de programación en C++, porque se frustra una de las ventajas de C++: la seguridad de tipos. A las funciones declaradas con puntos suspensivos se les aplican conversiones diferentes de las que se aplican a las funciones para las que se conocen los tipos de argumento formal y real:

- Si el argumento real es del tipo `float`, se promueve al tipo `double` antes de la llamada a función.
- Cualquier `signed char` o `unsigned char`, `signed short` o `unsigned short`, tipo de enumeración o campo de bits se convierte tanto en un `signed int` o `unsigned int` mediante la promoción integral.
- Cualquier argumento de tipo de clase se pasa por valor como estructura de datos; la copia se crea mediante copia binaria en lugar de invocar el constructor de copia de clase (si existe).

Los puntos suspensivos, si se usan, se deben declarar en último lugar en la lista de argumentos. Para obtener más información sobre cómo pasar un número variable de argumentos, vea la explicación de [va_arg](#), [va_start](#) y [va_list](#) en la *Referencia de la biblioteca en tiempo de ejecución*.

Para obtener información sobre los argumentos predeterminados en la programación CLR, consulte [Listas de argumentos variables \(...\) \(C++/CLI\)](#).

Los argumentos predeterminados permiten especificar el valor que un argumento debe asumir si no se proporciona ninguno en la llamada a función. El fragmento de código siguiente muestra cómo funcionan los argumentos predeterminados. Para obtener más información sobre las restricciones en la especificación de argumentos predeterminados, consulte [Argumentos predeterminados](#).

C++

```
// expre_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;
}

```

El programa anterior declara una función, `print`, que toma dos argumentos. Sin embargo, el segundo argumento `terminator`, tiene un valor predeterminado `"\n"`. En `main`, las dos primeras llamadas a `print` permiten que el segundo argumento predeterminado proporcione una nueva línea para finalizar la cadena impresa. La tercera llamada especifica un valor explícito para el segundo argumento. El resultado del programa es

Output

```

hello,
world!
good morning, sunshine.

```

Consulte también

[Tipos de expresiones](#)

Expresiones con operadores unarios

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Los operadores unarios actúan solo sobre un operando en una expresión. Los operadores unarios son los siguientes:

- Operador de direccionamiento indirecto (*)
- Operador address-of (&)
- Operador unario más (+)
- Operador unario de negación (-)
- Operador lógico de negación (!)
- Operador de complemento de uno (~)
- Operador de incremento prefijo (++)
- Operador de decremento de prefijo (--)
- Operador de conversión ()
- sizeof operador
- alignof operador
- Expresión
- new operador
- delete operador

Estos operadores tienen asociatividad de derecha a izquierda. Las expresiones unarias normalmente usan sintaxis que precede a una expresión de postfijo o primaria.

Sintaxis

`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: uno de
* & + - ! ~
```

Comentarios

Cualquier *postfix-expression* se considera una *unary-expression* y, dado que cualquier *primary-expression* se considera una *postfix-expression*, cualquier *primary-expression* también se considera una *unary-expression*. Para obtener más información, consulte [Expresiones postfijas](#) y [Expresiones primarias](#).

cast-expression es una *unary-expression* con una conversión opcional para cambiar el tipo. Para obtener más información, consulte [Operador de conversión \(\)](#).

noexcept-expression es un *noexcept-specifier* objeto con un argumento *constant-expression*. Para más información, consulte [noexcept](#).

new-expression hace referencia al operador [new](#). *delete-expression* hace referencia al operador [delete](#). Para obtener más información, consulte [Operador new](#) y [Operador delete](#).

Consulte también

[Tipos de expresiones](#)

Expresiones con operadores binarios

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Los operadores binarios actúan sobre dos operandos de una expresión. Los operadores binarios son:

- Operadores de multiplicación
 - Multiplicación (*)
 - División (/)
 - Módulo (%)
- Operadores aditivos
 - Suma (+)
 - Resta (-)
- Operadores de desplazamiento
 - Desplazamiento a la derecha (>>)
 - Desplazamiento a la izquierda (<<)
- Operadores relacionales y de igualdad
 - Menor que (<)
 - Mayor que (>)
 - Menor o igual que (<=)
 - Mayor o igual que (>=)
 - Igual a (==)
 - Distinto de (!=)
- Operadores bit a bit
 - AND bit a bit (&)
 - OR exclusivo bit a bit (^)
 - OR inclusivo bit a bit (|)

- Operadores lógicos
 - AND lógico (`&&`)
 - OR lógico (`||`)
- Operadores de asignación
 - Asignación (`=`)
 - Asignación y suma (`+ =`)
 - Asignación y resta (`- =`)
 - Asignación y multiplicación (`* =`)
 - Asignación y división (`/ =`)
 - Asignación y módulo (`% =`)
 - Asignación y desplazamiento a la izquierda (`<< =`)
 - Asignación y desplazamiento a la derecha (`>> =`)
 - Asignación AND bit a bit (`& =`)
 - Asignación y OR exclusivo bit a bit (`^ =`)
 - Asignación y OR inclusivo bit a bit (`| =`)
- Operador coma (,)

Consulte también

[Tipos de expresiones](#)

Expresiones constantes de C++

Artículo • 26/09/2022 • Tiempo de lectura: 2 minutos

Un valor *constante* es aquel que no cambia. C++ proporciona dos palabras clave para que pueda expresar la intención de que un objeto no está pensado para ser modificado y aplicar dicha intención.

C++ requiere expresiones constantes —expresiones que se evalúan como una constante— para las declaraciones de:

- Límites de matrices
- Selectores en instrucciones case
- Especificación de longitud de campo de bits
- Inicializadores de enumeración

Los únicos operandos que son válidos en expresiones constantes son:

- Literales
- Constantes de enumeración
- Valores declarados como const que se inicializan con expresiones constantes
- Expresiones `sizeof`

Las constantes no íntegras deben convertirse (explícita o implícitamente) en tipos enteros para ser válidos en una expresión constante. Por consiguiente, el código siguiente es legal.

C++

```
const double Size = 11.0;
char chArray[(int)Size];
```

Las conversiones explícitas a tipos enteros son legales en las expresiones constantes; el resto de tipos y los tipos derivados no son válidos, excepto si se usan como operandos del operador `sizeof`.

El operador de coma y los operadores de asignación no se pueden utilizar en expresiones constantes.

Consulte también

[Tipos de expresiones](#)

Semántica de las expresiones

Artículo • 03/03/2023 • Tiempo de lectura: 4 minutos

Las expresiones se evalúan según la prioridad y agrupación de sus operadores. (En [Prioridad y asociatividad de los operadores en Convenciones léxicas](#) se muestran las relaciones que los operadores de C++ imponen sobre las expresiones).

Orden de evaluación

Considere este ejemplo:

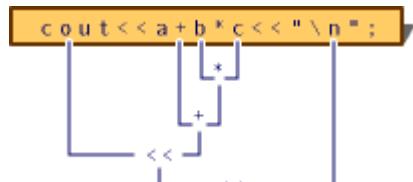
C++

```
// 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
```



Orden de expresión-evaluación

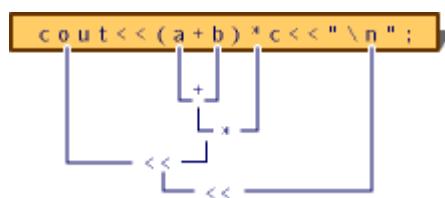
El orden en que se evalúa la expresión mostrada en la ilustración anterior viene determinado por la prioridad y la asociatividad de los operadores:

1. La multiplicación (*) tiene la prioridad más alta en esta expresión; por consiguiente, la subexpresión `b * c` se evalúa primero.

2. La suma (+) es la siguiente operación con mayor prioridad, por lo que `a` se suma al producto de `b` y `c`.

3. El desplazamiento a la izquierda (<<) tiene la prioridad más baja en la expresión, pero aparece dos veces. Como el operador de desplazamiento a la izquierda se agrupa de izquierda a derecha, la subexpresión de la izquierda se evalúa primero y después se evalúa la de la derecha.

Cuando se utilizan paréntesis para agrupar subexpresiones, se modifica la prioridad, así como el orden en que se evalúa la expresión, como se muestra en la ilustración siguiente.



Orden de expresión-evaluación con paréntesis

Las expresiones como las de la ilustración anterior se evalúan simplemente para que se apliquen sus efectos secundarios (en este caso, para transferir información al dispositivo de salida estándar).

Notación en expresiones

El lenguaje C++ indica ciertas compatibilidades al especificar operandos. En la tabla siguiente se muestran los tipos de operandos aceptables para los operadores que requieren operandos de tipo *type*.

Tipos de operando aceptables para los operadores

| Tipo esperado | Tipos permitidos |
|-------------------|---|
| <code>type</code> | <code>const type</code> <code>volatile type</code> <code>type&</code> <code>const type&</code> <code>volatile type&</code> <code>volatile const type</code> <code>volatile const type&</code> |

| Tipo esperado | Tipos permitidos |
|----------------------------|--|
| <code>type *</code> | <code>type *</code> <code>const type *</code> <code>volatile type *</code> <code>volatile const type *</code> |
| <code>const type</code> | <code>type</code> <code>const type</code> <code>const type&</code> |
| <code>volatile type</code> | <code>type</code> <code>volatile type</code> <code>volatile type&</code> |

Dado que las reglas anteriores se pueden utilizar combinadas, se puede proporcionar un puntero const a un objeto volatile cuando se espera un puntero.

Expresiones ambiguas

Algunas expresiones son ambiguas en su significado. Estas expresiones aparecen frecuentemente cuando el valor de un objeto se modifica más de una vez en la misma expresión. Estas expresiones se basan en un orden concreto de evaluación donde el lenguaje no define uno. Considere el ejemplo siguiente:

```
int i = 7;
func( i, ++i );
```

El lenguaje C++ no garantiza el orden en el que se evalúan los argumentos para una llamada de función. Por consiguiente, en el ejemplo anterior, `func` podría recibir los valores 7 y 8 u 8 y 8 para sus parámetros, dependiendo de si los parámetros se evaluaran de izquierda a derecha o de derecha a izquierda.

Puntos de secuencia de C++ (específicos de Microsoft)

Una expresión puede modificar el valor de un objeto solo una vez entre "puntos de secuencia" consecutivos.

La definición del lenguaje C++ no especifica actualmente puntos de secuencia. Microsoft C++ utiliza los mismos puntos de secuencia que ANSI C para cualquier expresión que contenga operadores de C y que no use operadores sobrecargados. Cuando los operadores están sobrecargados, la semántica cambia de la secuencia de operadores a la secuencia de llamadas de función. Microsoft C++ utiliza los puntos de secuencia siguientes:

- Operando izquierdo del operador AND lógico (`&&`). El operando izquierdo del operador AND lógico se evalúa totalmente y se aplican todos los efectos secundarios antes de continuar. No hay ninguna garantía de que se evalúe el operando derecho del operador AND lógico.
- Operando izquierdo del operador OR lógico (`||`). El operando izquierdo del operador OR lógico se evalúa totalmente y se aplican todos los efectos secundarios antes de continuar. No hay ninguna garantía de que se evalúe el operando derecho del operador OR lógico.
- Operando izquierdo del operador de coma. El operando izquierdo del operador de coma se evalúa totalmente y se aplican todos los efectos secundarios antes de continuar. Los dos operandos del operador de coma se evalúan siempre.
- Operador de llamada de función. La expresión de llamada de función y todos los argumentos de una función, incluidos los argumentos predeterminados, se evalúan y se aplican todos los efectos secundarios antes de que empiece la función. No hay ningún orden de evaluación específico entre los argumentos o la expresión de llamada de función.
- Primer operando del operador condicional. El primer operando del operador condicional se evalúa totalmente y se aplican todos los efectos secundarios antes de continuar.
- El final de una expresión de inicialización completa, como el final de una inicialización en una instrucción de declaración.
- La expresión de una instrucción de expresión. Las instrucciones de expresión constan de una expresión opcional seguida de un punto y coma (`;`). La expresión se evalúa completamente para aplicar sus efectos secundarios.
- La expresión de control en una instrucción de selección (if o switch). La expresión se evalúa completamente y se aplican todos los efectos secundarios antes de que se ejecute el código dependiente de la selección.
- La expresión de control de una instrucción while o do. La expresión se evalúa completamente y se aplican todos los efectos secundarios antes de que se ejecute

alguna instrucción de la siguiente iteración del bucle while o do.

- Cada una de las tres expresiones de una instrucción for. Cada expresión se evalúa completamente y se aplican todos los efectos secundarios antes de pasar a la siguiente expresión.
- La expresión de una instrucción return. La expresión se evalúa completamente y se aplican todos los efectos secundarios antes de devolver el control a la función de llamada.

Consulte también

[Expresiones](#)

Conversión

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

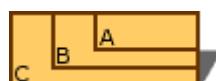
El lenguaje C++ permite que, si una clase se deriva de una clase base que contiene funciones virtuales, se pueda usar un puntero a ese tipo de clase base para llamar a las implementaciones de las funciones virtuales que residen en el objeto de la clase derivada. Una clase que contiene funciones virtuales se denomina a veces "clase polimórfica".

Como una clase derivada contiene en su totalidad las definiciones de todas las clases base de la que se deriva, es seguro convertir un puntero a la jerarquía de clases en cualquiera de estas clases base. Dado un puntero a una clase base, es seguro convertir el puntero en cualquier objeto que se encuentre por debajo en la jerarquía. Es seguro si el objeto al que se apunta es de un tipo derivado de la clase base. En este caso, se dice que el objeto real es el "objeto completo". Se dice que el puntero a la clase base apunta a un "subobjeto" del objeto completo. Considere, por ejemplo, la jerarquía de clases que se muestra en la ilustración siguiente.



Jerarquía de clases

Un objeto de tipo `C` se podría visualizar tal y como se muestra en la ilustración siguiente.



Clase C con subobjetos B y A

Dada una instancia de la clase `C`, hay un subobjeto `B` y un objeto `A`. La instancia de `C`, incluidos los subobjetos `A` y `B`, es el "objeto completo".

Mediante el uso de información de tipos en tiempo de ejecución, es posible determinar si un puntero apunta realmente a un objeto completo y si se puede realizar una conversión segura para que apunte a otro objeto de su jerarquía. El operador `dynamic_cast` se puede utilizar para realizar estos tipos de conversiones. También realiza la comprobación en tiempo de ejecución necesaria para que la operación sea segura.

Para la conversión de tipos que no son polimórficos, puede usar el operador `static_cast` (en este tema se explica la diferencia entre las conversiones estáticas y dinámicas, y cuándo es adecuado utilizar cada una de ellas).

En esta sección se describen los temas siguientes:

- [Operadores de conversión](#)
- [Información de tipos en tiempo de ejecución](#)

Consulte también

[Expresiones](#)

Operadores de conversión

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Hay varios operadores de conversión específicos del lenguaje C++. Estos operadores están diseñados para quitar una parte de la ambigüedad y riesgo inherentes a las conversiones antiguas del lenguaje C. Estos operadores son:

- [dynamic_cast](#) Se usa para la conversión de tipos polimórficos.
- [static_cast](#) Se usa para la conversión de tipos no polimórficos.
- [const_cast](#) Se usa para quitar los atributos `const`, `volatile` y `__unaligned`.
- [reinterpret_cast](#) Se usa para la reinterpretación simple de bits.
- [safe_cast](#) Se usa en C++/CLI para producir MSIL que se puede comprobar.

Use `const_cast` y `reinterpret_cast` como último recurso, ya que estos operadores plantean los mismos peligros que las conversiones antiguas. Sin embargo, siguen siendo necesarios para reemplazar completamente las conversiones antiguas.

Consulte también

[Conversión](#)

dynamic_cast (Operador)

Artículo • 03/03/2023 • Tiempo de lectura: 7 minutos

Convierte el operando `expression` en un objeto del tipo `type-id`.

Sintaxis

```
dynamic_cast < type-id > ( expression )
```

Comentarios

`type-id` debe ser un puntero o una referencia a un tipo de clase definido previamente o un "puntero a void". El tipo de `expression` debe ser un puntero si `type-id` es un puntero, o un valor L si `type-id` es una referencia.

Consulte [static_cast](#) para obtener una explicación de la diferencia entre las conversiones de fundición estática y dinámica, y cuándo es apropiado utilizar cada una.

Hay dos cambios de ruptura en el comportamiento del `dynamic_cast` en el código administrado:

- `dynamic_cast` a un puntero al tipo subyacente de una enumeración conversión boxing producirá un error en tiempo de ejecución, devolviendo 0 en lugar del puntero convertido.
- `dynamic_cast` ya no lanzará una excepción cuando `type-id` sea un puntero interior a un tipo de valor, fallando la conversión en tiempo de ejecución. La conversión devolverá ahora el valor de puntero 0 en lugar de producirse una excepción.

Si `type-id` es un puntero a una clase base directa o indirecta accesible de forma no ambigua desde `expression`, el resultado es un puntero al subobjeto único de tipo `type-id`. Por ejemplo:

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
}

```

Este tipo de conversión se denomina "conversión hacia arriba" porque sube un puntero en una jerarquía de clases, desde una clase derivada a una clase de la que se deriva. Una conversión hacia arriba es una conversión implícita.

Si `type-id` es `void*`, se realiza una comprobación en tiempo de ejecución para determinar el tipo real de `expression`. El resultado es un puntero al objeto completo al que apunta `expression`. Por ejemplo:

```

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
}

```

Si `type-id` no es `void*`, se realiza una comprobación en tiempo de ejecución para ver si el objeto al que apunta `expression` se puede convertir al tipo indicado por `type-id`.

Si el tipo de `expression` es una clase base del tipo de `type-id`, se realiza una comprobación en tiempo de ejecución para ver si `expression` apunta realmente a un objeto completo del tipo de `type-id`. Si es cierto, el resultado es un puntero a un objeto completo del tipo de `type-id`. Por ejemplo:

```

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
}

```

Este tipo de conversión se denomina "conversión hacia abajo" porque baja un puntero en una jerarquía de clases, desde una clase especificada a una clase derivada de ella.

En los casos de herencia múltiple, se introducen posibilidades de ambigüedad.

Considere la jerarquía de clases que se muestra en la ilustración siguiente.

Para los tipos CLR, `dynamic_cast` el resultado es un no-op si la conversión puede realizarse implícitamente, o una instrucción MSIL `isinst`, que realiza una comprobación dinámica y devuelve `nullptr` si la conversión falla.

El siguiente ejemplo utiliza `dynamic_cast` para determinar si una clase es una instancia de un tipo particular:

C++

```

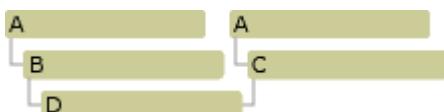
// dynamic_cast_clr.cpp
// compile with: /clr
using namespace System;

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);
}

```



Jerarquía de clases que muestra herencia múltiple

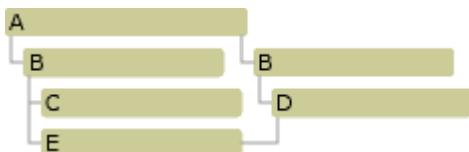
Un puntero a un objeto de tipo `D` se puede convertir de manera segura a `B` o a `C`. Sin embargo, si `D` se convierte para que apunte a un objeto `A`, ¿qué instancia de `A` resultaría? Esto produciría un error de conversión ambigua. Para eludir este problema, puede realizar dos conversiones no ambiguas. Por ejemplo:

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
}
```

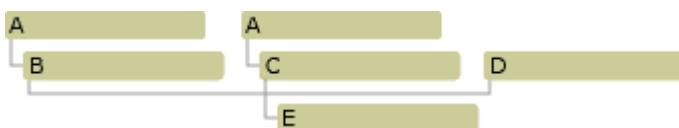
Se pueden introducir más ambigüedades cuando se utilizan clases base virtuales. Considere la jerarquía de clases que se muestra en la ilustración siguiente.



Jerarquía de clases que muestra clases base virtuales

En esta jerarquía, `A` es una clase base virtual. Dada una instancia de clase `E` y un puntero al `A` subobjeto, un `dynamic_cast` a un puntero a `B` fallará debido a la ambigüedad. Primero debe convertir de nuevo al objeto `E` completo y después volver a subir por la jerarquía, de forma no ambigua, hasta llegar al objeto `B` correcto.

Considere la jerarquía de clases que se muestra en la ilustración siguiente.



Jerarquía de clases que muestra clases base duplicadas

Dado un objeto de tipo `E` y un puntero al subobjeto `D`, para navegar desde el subobjeto `D` hasta el subobjeto `A` situado más a la izquierda, se pueden realizar tres conversiones. Se puede realizar una conversión `dynamic_cast` del puntero `D` a un

puntero `E`, luego una conversión (cualquier `dynamic_cast` o una conversión implícita) de `E` a `B`, y finalmente una conversión implícita de `B` a `A`. Por ejemplo:

```
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  
}
```

El operador `dynamic_cast` también puede utilizarse para realizar un "conversión cruzada". Utilizando la misma jerarquía de clases, es posible convertir un puntero, por ejemplo, del subobjeto `B` al subobjeto `D`, siempre que el objeto completo sea de tipo `E`.

Teniendo en cuenta las conversiones cruzadas, en realidad es posible efectuar la conversión de un puntero a `D` a un puntero al subobjeto `A` situado más a la izquierda en solo dos pasos. Puede realizar una conversión cruzada de `D` a `B` y después una conversión implícita de `B` a `A`. Por ejemplo:

```
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  
}
```

Un valor de puntero nulo se convierte en el valor de puntero nulo del tipo de destino mediante `dynamic_cast`.

Cuando se utiliza `dynamic_cast < type-id > (expression)`, si `expression` no se puede convertir de forma segura al tipo `type-id`, la comprobación en tiempo de ejecución hace que se produzca un error en la conversión. Por ejemplo:

```
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  
}
```

El valor de una conversión con errores al tipo de puntero es el puntero NULL. Una conversión fallida a un tipo de referencia lanza una [Excepción bad_cast](#). Si `expression` no apunta o hace referencia a un objeto válido, se lanza una excepción `_non_rtti_object`.

Consulte [typeid](#) para una explicación de la excepción `_non_rtti_object`.

Ejemplo

En el ejemplo siguiente se crea el puntero de la clase base (struct A) a un objeto (struct C). Esto, además del hecho de que hay funciones virtuales, hace posible el polimorfismo en tiempo de ejecución.

En el ejemplo también se llama a una función no virtual de la jerarquía.

```
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);

    // will fail because B knows nothing about C
    B BonStack;
    Globaltest(BonStack);
}

```

Output

```
in C
test2 in B
in GlobalTest
Can't cast to C
```

Consulte también

[Operadores de conversión](#)

[Palabras clave](#)

bad_cast (Excepción)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

El operador `dynamic_cast` inicia la excepción `bad_cast` como resultado de una conversión incorrecta a un tipo de referencia.

Sintaxis

```
catch (bad_cast)
    statement
```

Comentarios

La interfaz para `bad_cast` es:

```
C++

class bad_cast : public exception
```

El código siguiente contiene un ejemplo de `dynamic_cast` con errores que inicia la excepción `bad_cast`.

```
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();
}

```

Se inicia una excepción porque el objeto que se convierte (una forma) no se deriva del tipo de conversión especificado (círculo). Para evitar la excepción, agregue estas declaraciones a `main`:

C++

```

Circle circle_instance;
Circle& ref_circle = circle_instance;

```

A continuación, invierta el sentido de la conversión en el bloque `try` de la siguiente manera:

C++

```

Shape& ref_shape = dynamic_cast<Shape&>(ref_circle);

```

Miembros

Constructores

| Constructor | Descripción |
|-----------------------|--|
| <code>bad_cast</code> | Constructor para los objetos de tipo <code>bad_cast</code> . |

Functions

| Función | Descripción |
|------------------|-------------|
| <code>qué</code> | TBD |

Operadores

| Operador | Descripción |
|---------------------------|---|
| operator= | Operador de asignación que asigna un objeto <code>bad_cast</code> a otro. |

bad_cast

Constructor para los objetos de tipo `bad_cast`.

C++

```
bad_cast(const char * _Message = "bad cast");
bad_cast(const bad_cast &);
```

operator=

Operador de asignación que asigna un objeto `bad_cast` a otro.

C++

```
bad_cast& operator=(const bad_cast&) noexcept;
```

what

C++

```
const char* what() const noexcept override;
```

Consulte también

[dynamic_cast \(Operador\)](#)

[Palabras clave](#)

[Procedimientos C++ recomendados modernos para excepciones y control de errores](#)

static_cast (Operador)

Artículo • 03/03/2023 • Tiempo de lectura: 4 minutos

Convierte una *expresión* al tipo de *type-id*, basándose únicamente en los tipos que se encuentran en la expresión.

Sintaxis

```
static_cast <type-id> ( expression )
```

Comentarios

En Standard C++, no se realiza ninguna comprobación de tipo en tiempo de ejecución como ayuda para garantizar la seguridad de la conversión. En C++/CX, se realiza una comprobación en tiempo de compilación y en tiempo de ejecución. Para obtener más información, consulta [Conversión](#).

Se puede usar el operador `static_cast` para las operaciones como convertir un puntero a una clase base en un puntero a una clase derivada. Estas conversiones no siempre son seguras.

En general, debe usar `static_cast` cuando desee convertir tipos de datos numéricos como enumeraciones en valores de tipo int o valores de tipo int en flotantes, y sepa con seguridad los tipos de datos implicados en la conversión. Las conversiones `static_cast` no son tan seguras como las conversiones `dynamic_cast`, ya que `static_cast` no realiza ninguna comprobación de tipo en tiempo de ejecución, mientras que `dynamic_cast` sí la hace. Una conversión `dynamic_cast` a un puntero ambiguo producirá un error, mientras que `static_cast` vuelve como si no hubiera nada incorrecto; esto puede ser peligroso. Aunque las conversiones `dynamic_cast` son más seguras, `dynamic_cast` solo funciona en punteros o referencias, y la comprobación del tipo en tiempo de ejecución supone una sobrecarga. Para obtener más información, consulte el [Operador dynamic_cast](#).

En el ejemplo siguiente, la línea `D* pd2 = static_cast<D*>(pb);` no es segura porque `D` puede tener campos y métodos que no están en `B`. Sin embargo, la línea `B* pb2 = static_cast<B*>(pd);` es una conversión segura porque `D` siempre contiene todo `B`.

```

// 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.
}

```

A diferencia de `dynamic_cast`, no se realiza ninguna comprobación en tiempo de ejecución en la conversión `static_cast` de `pb`. El objeto al que apunta `pb` puede no ser un objeto de tipo `D`, en cuyo caso el uso de `*pd2` podría ser desastroso. Por ejemplo, la llamada a una función que es miembro de la clase `D`, pero no de la clase `B`, podría producir una infracción de acceso.

Los operadores `dynamic_cast` y `static_cast` mueven un puntero en una jerarquía de clases. Sin embargo, `static_cast` usa exclusivamente la información proporcionada en la instrucción de conversión y, por tanto, puede que no sea segura. Por ejemplo:

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);
}

```

Si `pb` apunta realmente un objeto de tipo `D`, `pd1` y `pd2` obtienen el mismo valor.

También obtienen el mismo valor si `pb == 0`.

Si `pb` apunta a un objeto de tipo `B` y no a toda la clase `D`, `dynamic_cast` sabe suficiente para devolver cero. Sin embargo, `static_cast` usa la aserción del programador de que `pb` apunta a un objeto de tipo `D` y simplemente devuelve un puntero a ese objeto `D` supuesto.

Por tanto, `static_cast` puede hacer lo contrario de las conversiones implícitas, en cuyo caso los resultados son indefinidos. Es el programador quien tiene que comprobar que los resultados de una conversión `static_cast` son seguros.

Este comportamiento también se aplica a los tipos distintos de los tipos de clase. Por ejemplo, se puede usar `static_cast` para convertir de un tipo `int` a un tipo `char`. Sin embargo, el tipo `char` resultante quizás no tenga suficientes bits para almacenar todo el valor `int`. Una vez más, es el programador quien tiene que comprobar que los resultados de una conversión `static_cast` son seguros.

También se puede utilizar el operador `static_cast` para realizar cualquier conversión implícita, incluidas las conversiones estándar y las conversiones definidas por el usuario. Por ejemplo:

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);
}
```

El operador `static_cast` puede convertir explícitamente un valor entero en un tipo de enumeración. Si el valor del tipo entero no está dentro del intervalo de valores de enumeración, el valor de enumeración resultante no está definido.

El operador `static_cast` convierte un valor de puntero NULL al valor de puntero NULL del tipo de destino.

El operador `static_cast` puede convertir explícitamente cualquier expresión al tipo `void`. El tipo `void` de destino puede incluir opcionalmente el atributo `const`, `volatile` o `_unaligned`.

El operador `static_cast` no puede desechar los atributos `const`, `volatile` o `_unaligned`. Consulte el [Operador `const_cast`](#) para obtener información sobre cómo quitar estos atributos.

C++/CLI: Debido al riesgo que supone realizar conversiones no comprobadas además de reubicar un recolector de elementos no utilizados, `static_cast` solo debe utilizarse en código que sea crítico para el rendimiento cuando esté seguro de que funcionará correctamente. Si debe usar `static_cast` en modo de lanzamiento, sustitúyalo con `safe_cast` en las compilaciones de depuración para asegurarse de que funciona correctamente.

Consulte también

[Operadores de conversión](#)

[Palabras clave](#)

const_cast (Operador)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Quita los atributos `const`, `volatile` y `__unaligned` de una clase.

Sintaxis

```
const_cast <type-id> (expression)
```

Comentarios

Un puntero a cualquier tipo de objeto o un puntero a un miembro de datos se puede convertir explícitamente a un tipo que sea idéntico, salvo en el caso de los calificadores `const`, `volatile` y `__unaligned`. Para los punteros y las referencias, el resultado hará referencia al objeto original. Para los punteros a miembros de datos, el resultado hará referencia al mismo miembro que el puntero original (sin convertir) al miembro de datos. Según el tipo del objeto al que se hace referencia, una operación de escritura a través del puntero, referencia o puntero a miembro de datos resultante podría generar un comportamiento indefinido.

No puede utilizar el operador `const_cast` para invalidar directamente el estado constante de una variable constante.

El operador `const_cast` convierte un valor de puntero nulo al valor de puntero nulo del tipo de destino.

Ejemplo

C++

```
// expte_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();  
}
```

En la línea que contiene `const_cast`, el tipo de datos del puntero `this` es `const CCTest *`. El operador `const_cast` cambia el tipo de datos del puntero `this` a `CCTest *`, lo que permite que se modifique el elemento `number` miembro. La conversión se produce únicamente para el resto de la instrucción en la que aparece.

Consulte también

[Operadores de conversión](#)

[Palabras clave](#)

reinterpret_cast (Operador)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Permite que cualquier puntero se convierta en cualquier otro tipo de puntero. También permite convertir cualquier tipo entero en cualquier tipo de puntero y viceversa.

Sintaxis

```
reinterpret_cast < type-id > ( expression )
```

Comentarios

El uso incorrecto del operador `reinterpret_cast` puede ser no seguro. A menos que la conversión deseada sea inherentemente de bajo nivel, se debe utilizar uno de los otros operadores de conversión.

El operador `reinterpret_cast` se puede utilizar para las conversiones como `char*` a `int*` o `One_class*` a `Unrelated_class*`, que son intrínsecamente no seguras.

El resultado de `reinterpret_cast` no se puede utilizar con seguridad para algo distinto que convertirlo otra vez a su tipo original. Otros usos son, en el mejor de los casos, no portables.

El operador `reinterpret_cast` no puede desechar los atributos `const`, `volatile` o `_unaligned`. Consulte [const_cast \(operador\)](#) para obtener información sobre cómo quitar estos atributos.

El operador `reinterpret_cast` convierte un valor de puntero nulo al valor de puntero nulo del tipo de destino.

Un uso práctico de `reinterpret_cast` es el que se hace en una función hash, que asigna un valor a un índice de tal forma que dos valores distintos raramente acaben teniendo el mismo índice.

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
```

`reinterpret_cast` permite que el puntero se tratará como tipo entero. El resultado se cambia a bits y se compara mediante XOR consigo mismo para producir un índice único (con un alto grado de probabilidad). El índice se trunca mediante una conversión de estilo de C al tipo de valor devuelto de la función.

Consulte también

[Operadores de conversión](#)

[Palabras clave](#)

Información de tipos en tiempo de ejecución

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

La información de tipo en tiempo de ejecución (RTTI) es un mecanismo que permite determinar el tipo de un objeto durante la ejecución del programa. RTTI se agregó al lenguaje C++ porque muchos proveedores de bibliotecas de clases implementaban esta funcionalidad por sí mismos. Esto produjo incompatibilidades entre las bibliotecas. Por tanto, se hizo obvio que se necesitaba compatibilidad para información de tipo en tiempo de ejecución en el nivel de lenguaje.

Por razones de claridad, esta discusión de RTTI se limita casi totalmente a punteros. Sin embargo, los conceptos discutidos también se aplican a referencias.

Hay tres elementos principales del lenguaje C++ para la información en tiempo de ejecución:

- El operador `dynamic_cast`.

Se usa para la conversión de tipos polimórficos.

- El operador `typeid`.

Se utiliza para identificar el tipo exacto de un objeto.

- La clase `type_info`.

Se usa para contener la información de tipo devuelta por el operador `typeid`.

Consulte también

[Conversión](#)

bad_typeid (Excepción)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

La excepción **bad_typeid** la produce el [operador typeid](#) cuando el operando de `typeid` es un puntero NULL.

Sintaxis

```
catch (bad_typeid)
    statement
```

Comentarios

La interfaz para **bad_typeid** es:

```
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;
};
```

En el ejemplo siguiente se muestra el operador `typeid` que produce una excepción **bad_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;
}
}
```

Salida

Output

Object is NULL

Vea también

[Información de tipos en tiempo de ejecución](#)

[Palabras clave](#)

type_info (Clase)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

La clase `type_info` describe la información de tipo generada en el programa por el compilador. Los objetos de esta clase almacenan de forma eficaz un puntero a un nombre para el tipo. La clase `type_info` también almacena un valor codificado adecuado para comparar dos tipos en cuanto a igualdad u orden de intercalación. Las reglas de codificación y la secuencia de intercalación para tipos no se especifican y pueden diferir entre programas.

El archivo de encabezado `<typeinfo>` debe incluirse para utilizar la clase `type_info`. La interfaz para la clase `type_info` es:

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;
};
```

No puede crear instancias de objetos de la clase `type_info` directamente porque la clase solo tiene un constructor de copias privado. La única manera de crear un objeto `type_info` (temporal) es utilizar el operador `typeid`. Como el operador de asignación también es privado, no puede copiar ni asignar objetos de la clase `type_info`.

`type_info::hash_code` define una función hash adecuada para asignar valores de tipo `typeinfo` a una distribución de valores de índice.

Los operadores `==` y `!=` se pueden utilizar para comparar la igualdad y la desigualdad con otros objetos `type_info`, respectivamente.

No hay ningún vínculo entre el orden de intercalación de tipos y las relaciones de herencia. Utilice la función miembro `type_info::before` para determinar la secuencia de intercalación de tipos. No hay ninguna garantía de que `type_info::before` produzca el

mismo resultado en programas diferentes o incluso ejecuciones diferentes del mismo programa. De esta manera, `type_info::before` es similar al operador (`&`) address-of.

La función miembro `type_info::name` devuelve `const char*` a una cadena finalizada en null que representa el nombre legible del tipo. La memoria a la que se señala se almacena en caché y nunca debe desasignarse directamente.

La función miembro `type_info::raw_name` devuelve `const char*` a una cadena finalizada en null que representa el nombre representativo del tipo de objeto. El nombre se almacena realmente en forma representativa para ahorrar espacio. Por consiguiente, esta función es más rápida que `type_info::name` porque no necesita anular la aplicación de nombre representativo. La cadena devuelta por la función `type_info::raw_name` es útil en operaciones de comparación pero no es legible. Si necesita una cadena legible, utilice en su lugar la función `type_info::name`.

La información de tipo se genera para las clases polimórficas solo si se especifica la opción del compilador [/GR \(Habilitar información de tipo en tiempo de ejecución\)](#).

Vea también

[Información de tipos en tiempo de ejecución](#)

Instrucciones (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Las instrucciones de C++ son los elementos de programa que controlan cómo y en qué orden se manipulan los objetos. Esta sección incluye:

- [Información general](#)
- [Instrucciones con etiqueta](#)
- Categorías de instrucciones
 - [Instrucciones de expresión](#). Estas instrucciones evalúan una expresión para ver sus efectos secundarios o para averiguar su valor devuelto.
 - [Instrucciones Null](#). Estas instrucciones se pueden proporcionar cuando la sintaxis de C++ requiere una instrucción pero no se va a realizar ninguna acción.
 - [Instrucciones compuestas](#). Estas instrucciones son grupos de instrucciones entre llaves ({}). Se pueden utilizar donde se puede utilizar una sola instrucción.
 - [Instrucciones de selección](#). Estas instrucciones realizan una prueba; a continuación, ejecutan una sección de código si la prueba se evalúa como true (distinto de cero). Pueden ejecutar otra sección de código si la prueba se evalúa como false.
 - [Instrucciones de iteración](#). Estas instrucciones ejecutan repetidamente un bloque de código hasta que se cumple un criterio de finalización especificado.
 - [Instrucciones de salto](#). Estas instrucciones transfieren el control inmediatamente a otra ubicación de la función o devuelven el control de la función.
 - [Instrucciones de declaración](#). Las declaraciones introducen un nombre en un programa.

Para obtener información sobre las instrucciones de control de excepciones, vea [Control de excepciones](#).

Vea también

[Referencia del lenguaje C++](#)

Información general sobre las instrucciones de C++

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Las instrucciones de C++ se ejecutan secuencialmente, excepto cuando una instrucción de expresión, una instrucción de selección, una instrucción de iteración o una instrucción de salto modifica específicamente esa secuencia.

Las instrucciones pueden ser de los tipos siguientes:

labeled-statement
expression-statement
compound-statement
selection-statement
iteration-statement
jump-statement
declaration-statement
try-throw-catch

En la mayoría de los casos, la sintaxis de la instrucción de C++ es idéntica a la de ANSI C89. La principal diferencia entre los dos es que en C89 las declaraciones solo se permiten al principio de un bloque; C++ agrega el elemento *declaration-statement*, que elimina eficazmente esta restricción. Esto permite introducir variables en un punto del programa donde se puede calcular un valor de inicialización precalculado.

Declarar variables dentro de bloques también permite controlar con precisión el ámbito y la duración de esas variables.

Los artículos sobre instrucciones describen las siguientes palabras clave de C++:

`break`
`case`
`catch`
`continue`
`default`
`do`
`else`
`_except`

```
_finally  
for  
goto  
  
if  
_if_exists  
_if_not_exists  
_leave  
return  
  
switch  
throw  
_try  
try  
while
```

Vea también

[Instrucciones](#)

Instrucciones con etiqueta

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Las etiquetas se usan para transferir el control de programas directamente a la instrucción especificada.

Sintaxis

```
Labeled-statement:  
    identifier : statement  
    case constant-expression : statement  
    default : statement
```

El ámbito de una etiqueta es toda la función donde se declara.

Comentarios

Hay tres tipos de instrucciones con etiquetas. En todas ellas se utiliza el carácter de dos puntos (:) para distinguir el tipo de etiqueta de la instrucción. Las etiquetas `case` y `default` son específicas para las instrucciones `case`.

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  
}
```

Etiquetas y la instrucción `goto`

La aparición de una etiqueta `identifier` en el programa de origen declara una etiqueta. Solo una instrucción `goto` puede transferir el control a una etiqueta `identifier`. En el siguiente fragmento de código se muestra el uso de la instrucción `goto` y una etiqueta `identifier`:

Una etiqueta no puede aparecer por sí misma; debe estar asociada siempre a una instrucción. Si se necesita la propia etiqueta, coloque una instrucción null detrás de la etiqueta.

La etiqueta tiene ámbito de función y no se puede volver a declarar dentro de la función. Sin embargo, se puede utilizar el mismo nombre como una etiqueta en diferentes funciones.

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.
```

Etiquetas en la instrucción `case`

Las etiquetas que aparecen después de la palabra clave `case` no pueden aparecer también fuera de una instrucción `switch`. (Esta restricción también se aplica a la palabra clave `default`). En el fragmento de código siguiente se muestra el uso correcto de las etiquetas `case`:

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;
}
```

Consulte también

[Información general sobre las instrucciones de C++](#)
[Instrucción switch \(C++\)](#)

Expression (Instrucción)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Las instrucciones de expresión hacen que se evalúen las expresiones. No se realiza ninguna transferencia de control o iteración como resultado de una instrucción de expresión.

La sintaxis de la instrucción de expresión es simplemente

Sintaxis

```
[expression] ;
```

Comentarios

Todas las expresiones de una instrucción de expresión se evalúan y se aplican todos los efectos secundarios antes de que se ejecute la siguiente instrucción. Las instrucciones de expresión más comunes son las asignaciones y las llamadas a funciones. Puesto que la expresión es opcional, un punto y coma solo se considera una instrucción de expresión vacía, denominada instrucción [null](#).

Consulte también

[Información general sobre las instrucciones de C++](#)

NULL (Instrucción)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

La "instrucción null" es una instrucción de expresión a la que le falta la *expresión*. Es útil cuando la sintaxis del lenguaje llama a una instrucción pero no a una evaluación de la expresión. Consta de un punto y coma.

Las instrucciones null se utilizan normalmente como marcadores de posición en instrucciones de iteración o como instrucciones en las que se colocan etiquetas al final de las instrucciones compuestas o funciones.

El siguiente fragmento de código muestra cómo copiar una cadena a otra e incorpora la instrucción null:

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()
{}
```

Consulte también

[Expression \(Instrucción\)](#)

Instrucciones compuestas (Bloques)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Una instrucción compuesta consta de cero o más instrucciones entre llaves ({}). Una instrucción compuesta se puede utilizar en cualquier lugar donde se espere una instrucción. Las instrucciones compuestas normalmente se denominan "bloques".

Sintaxis

```
{ [ statement-list ] }
```

Comentarios

En el ejemplo siguiente se utiliza una instrucción compuesta como la parte *statement* de la instrucción `if` (consulte [Instrucción if](#) para obtener más detalles sobre la sintaxis):

```
C++  
  
if( Amount > 100 )  
{  
    cout << "Amount was too large to handle\n";  
    Alert();  
}  
else  
{  
    Balance -= Amount;  
}
```

ⓘ Nota

Dado que una declaración es una instrucción, una declaración puede ser una de las instrucciones de *statement-list*. Por consiguiente, los nombres declarados dentro de una instrucción compuesta, pero no declarados explícitamente como static, tienen ámbito local y (para objetos) duración. Consulte [Ámbito](#) para obtener más información sobre el tratamiento de los nombres con ámbito local.

Consulte también

Información general sobre las instrucciones de C++

Instrucciones de selección (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Las instrucciones de selección de C++, [if](#) y [switch](#), proporcionan un medio de ejecutar secciones de código de forma condicional.

Las instrucciones [_if_exists](#) y [_if_not_exists](#) permiten incluir de forma condicional código dependiendo de la existencia de un símbolo.

Vea en cada tema individual la sintaxis de cada instrucción.

Consulte también

[Información general sobre las instrucciones de C++](#)

if-else (instrucción) (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

Una instrucción if-else controla la bifurcación condicional. Las instrucciones de `if-branch` se ejecutan solo si se `condition` evalúa como un valor distinto de cero (o `true`). Si el valor de `condition` es distinto de cero, se ejecuta la siguiente instrucción y se omite la instrucción que sigue a la instrucción `else` opcional. De lo contrario, se omite la siguiente instrucción y, si hay una instrucción `else`, la instrucción siguiente `else` se ejecuta.

Las expresiones `condition` que se evalúan como no cero son:

- `true`
- un puntero distinto de null,
- cualquier valor aritmético distinto de cero, o
- un tipo de clase que defina una conversión no ambigua a un tipo aritmético o puntero. (Para más información sobre las conversiones, consulte [Conversiones estándar](#).)

Sintaxis

`init-statement`:

`expression-statement`

`simple-declaration`

`condition`:

`expression`

`attribute-specifier-seqopt` `decl-specifier-seq` `declarator brace-or-equal-initializer`

`statement`:

`expression-statement`

`compound-statement`

`expression-statement`:

`expressionopt ;`

`compound-statement`:

`{ statement-seqopt }`

```

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 Este elemento opcional está disponible a partir de C++17.

if...else (instrucciones)

En ambos formatos de la instrucción `if`, se evalúa `condition`, que puede tener cualquier valor excepto una estructura, incluidos todos los efectos secundarios. El control pasa de la instrucción `if` a la siguiente instrucción del programa a menos que el `if-branch` o `else-branch` ejecutados contengan un `break`, `continue` o `goto`.

La cláusula `else` de una instrucción `if...else` está asociada a la instrucción `if` anterior más cercana del mismo ámbito que no tenga una instrucción `else` correspondiente.

Ejemplo

Este código de ejemplo muestra varias instrucciones `if` en uso, tanto con como sin `else`:

```

C++

// if_else_statement.cpp
#include <iostream>

using namespace std;

class C
{
public:
    void do_something(){}
};

void init(C){}

```

```

bool is_true() { return true; }
int x = 10;

int main()
{
    if (is_true())
    {
        cout << "b is true!\n"; // executed
    }
    else
    {
        cout << "b is false!\n";
    }

    // no else statement
    if (x == 10)
    {
        x = 0;
    }

    C* c;
    init(c);
    if (c)
    {
        c->do_something();
    }
    else
    {
        cout << "c is null!\n";
    }
}

```

Instrucción if con un inicializador

A partir de C++17, una instrucción `if` también puede contener una expresión `init-statement` que declara e inicializa una variable con nombre. Use esta forma de la instrucción if cuando la variable solo sea necesaria dentro del ámbito de la instrucción if. **Específico de Microsoft:** este formulario está disponible a partir de la versión 15.3 de Visual Studio 2017 y requiere al menos la opción del compilador `/std:c++17`.

Ejemplo

C++

```

#include <iostream>
#include <mutex>
#include <map>
#include <string>
#include <algorithm>

```

```

using namespace std;

map<int, string> m;
mutex mx;
bool shared_flag; // guarded by mx
void unsafe_operation() {}

int main()
{
    if (auto it = m.find(10); it != m.end())
    {
        cout << it->second;
        return 0;
    }

    if (char buf[10]; fgets(buf, 10, stdin))
    {
        m[0] += buf;
    }

    if (lock_guard<mutex> lock(mx); shared_flag)
    {
        unsafe_operation();
        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";
    }
}

```

Instrucciones if constexpr

A partir de C++17, puede usar una instrucción `if constexpr` en plantillas de función para tomar decisiones de bifurcación en tiempo de compilación sin tener que recurrir a varias sobrecargas de función. **Específico de Microsoft:** este formulario está disponible a partir de la versión 15.3 de Visual Studio 2017 y requiere al menos la opción del compilador `/std:c++17`.

Ejemplo

En este ejemplo se muestra cómo se puede escribir una sola función que controla el desempaquetado de parámetros. No se necesita ninguna sobrecarga de parámetro

cero:

C++

```
template <class T, class... Rest>
void f(T&& t, Rest&&... r)
{
    // handle t
    do_something(t);

    // handle r conditionally
    if constexpr (sizeof...(r))
    {
        f(r...);
    }
    else
    {
        g(r...);
    }
}
```

Consulte también

[Instrucciones de selección](#)

[Palabras clave](#)

[Instrucción switch \(C++\)](#)

`__if_exists` (Instrucción)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

La instrucción `__if_exists` prueba si existe el identificador especificado. Si el identificador existe, se ejecuta el bloque de instrucción especificado.

Sintaxis

```
__if_exists ( identifier ) {  
    statements  
};
```

Parámetros

identifier

El identificador cuya existencia se desea probar.

statements

Una o más instrucciones que se tienen que ejecutar si existe *identifier*.

Observaciones

⊗ Precaución

Para obtener los resultados más confiables, conviene usar la instrucción

`__if_exists` con las restricciones siguientes.

- Aplique la instrucción `__if_exists` solo a tipos simples y no a plantillas.
- Aplique la instrucción `__if_exists` a identificadores tanto dentro como fuera de una clase. No aplique la instrucción `__if_exists` a variables locales.
- Use la instrucción `__if_exists` solo en el cuerpo de una función. Fuera del cuerpo de una función, la instrucción `__if_exists` solo puede probar tipos totalmente definidos.

- Cuando se prueban funciones sobrecargadas, no se puede probar una forma específica de la sobrecarga.

El complemento a la instrucción `__if_exists` es la instrucción `__if_not_exists`.

Ejemplo

Observe que este ejemplo utiliza plantillas, lo que no se recomienda.

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;
}
```

Salida

Output

```
In X<T>::Dump()
In A::Dump()
In X<T>::Dump()
T::Dump does not exist
g_bFlag = 1
C::f exists
```

Consulte también

[Instrucciones de selección](#)

[Palabras clave](#)

[__if_not_exists \(Instrucción\)](#)

`__if_not_exists` (Instrucción)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

La instrucción `__if_not_exists` prueba si existe el identificador especificado. Si el identificador no existe, se ejecuta el bloque de instrucción especificado.

Sintaxis

```
__if_not_exists ( identifier ) {  
    statements  
};
```

Parámetros

identifier

El identificador cuya existencia se desea probar.

statements

Una o más instrucciones que se tienen que ejecutar si *identifier* no existe.

Observaciones

⊗ Precaución

Para obtener los resultados más confiables, conviene usar la instrucción

`__if_not_exists` con las restricciones siguientes.

- Aplique la instrucción `__if_not_exists` solo a tipos simples y no a plantillas.
- Aplique la instrucción `__if_not_exists` a identificadores tanto dentro como fuera de una clase. No aplique la instrucción `__if_not_exists` a variables locales.
- Use la instrucción `__if_not_exists` solo en el cuerpo de una función. Fuera del cuerpo de una función, la instrucción `__if_not_exists` solo puede probar tipos totalmente definidos.

- Cuando se prueban funciones sobrecargadas, no se puede probar una forma específica de la sobrecarga.

El complemento a la instrucción `_if_not_exists` es la instrucción [`_if_exists`](#).

Ejemplo

Para obtener un ejemplo sobre cómo usar `_if_not_exists`, consulte la [instrucción `_if_exists`](#).

Consulte también

[Instrucciones de selección](#)

[Palabras clave](#)

[_if_exists \(Instrucción\)](#)

Instrucción `switch` (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 4 minutos

Permite la selección entre varias secciones de código, dependiendo del valor de una expresión entera.

Sintaxis

selection-statement:

`switch (init-statementoptC++17 condition) statement`

init-statement:

`expression-statement`
`simple-declaration`

condition:

`expression`
`attribute-specifier-seqopt decl-specifier-seq declarator brace-or-equal-initializer`

Labeled-statement:

`case constant-expression : statement`
`default : statement`

Comentarios

Una instrucción `switch` hace que el control se transfiera a una instrucción *Labeled-statement* en el cuerpo de la instrucción, en función del valor de *condition*.

condition debe tener un tipo entero o ser de un tipo de clase que tiene una conversión no ambigua a un tipo entero. La promoción integral tiene lugar como se describe en [Conversiones estándar](#).

El cuerpo de la instrucción `switch` consta de una serie de etiquetas `case` y una etiqueta optional `default`. *Labeled-statement* es una de estas etiquetas y las instrucciones que siguen. Las instrucciones con etiquetas no son requisitos sintácticos, pero la instrucción `switch` no tiene sentido sin ellas. No puede haber dos valores *constant-expression* en

las instrucciones `case` que se evalúen en el mismo valor. La etiqueta `default` puede aparecer solo una vez. La instrucción `default` se coloca a menudo al final, pero puede aparecer en cualquier parte del cuerpo de la instrucción `switch`. Una etiqueta `case` o `default` solo puede aparecer en una instrucción `switch`.

El elemento `constant-expression` de cada etiqueta `case` se convierte en un valor constante que tiene el mismo tipo que `condition`. Después, se compara con `condition` para ver si son iguales. El control pasa a la primera instrucción después del valor `constant-expression` de `case` que coincide con el valor de `condition`. El comportamiento resultante se muestra en la siguiente tabla.

Comportamiento de la instrucción `switch`

| Condición | Acción |
|---|--|
| El valor convertido coincide con el de la expresión de control promovida. | El control se transfiere a la instrucción que sigue a esa etiqueta. |
| Ninguna de las constantes coincide con las constantes de las etiquetas <code>case</code> ; hay una etiqueta <code>default</code> . | El control se transfiere a la etiqueta <code>default</code> . |
| Ninguna de las constantes coincide con las constantes de las etiquetas <code>case</code> ; no hay una etiqueta <code>default</code> . | El control se transfiere a la instrucción situada detrás de la instrucción <code>switch</code> . |

Si se encuentra una expresión coincidente, la ejecución puede continuar por etiquetas `case` o `default` posteriores. La instrucción `break` se usa para detener la ejecución y transferir el control a la instrucción situada detrás de la instrucción `switch`. Sin una instrucción `break`, se ejecutan todas las instrucciones que hay desde la etiqueta `case` coincidente hasta el final de la instrucción `switch`, incluida la etiqueta `default`. Por ejemplo:

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) );
}

```

En el ejemplo anterior, `uppercase_A` se incrementa si `c` es una mayúscula `'A'`. La instrucción `break` detrás de `uppercase_A++` finaliza la ejecución del cuerpo de la instrucción `switch` y el control pasa al bucle `while`. Sin la instrucción `break`, la ejecución pasaría a la siguiente instrucción con etiqueta, de modo que `lowercase_a` y `other` también se incrementarían. La instrucción `break` para `case 'a'` tiene un propósito similar. Si `c` es una minúscula `'a'`, `lowercase_a` se incrementa y la `break` instrucción finaliza `switch` el cuerpo de la instrucción. Si `c` no es `'a'` ni `'A'`, se ejecuta la instrucción `default`.

Visual Studio 2017 y versiones posteriores (disponible en el modo `/std:c++17` y versiones posteriores): el atributo `[[fallthrough]]` se especifica en el estándar de C++17. Puede usarse en una instrucción `switch`. Es una sugerencia para el compilador (o para cualquier persona que lea el código) de que el comportamiento de paso explícito es intencionado. Actualmente, el compilador de Microsoft C++ no advierte sobre el comportamiento de paso explícito, por lo que este atributo no tiene ningún efecto sobre el comportamiento del compilador. En el ejemplo, el atributo se aplica a una instrucción vacía dentro de la instrucción etiquetada sin terminar. En otras palabras, el punto y coma es necesario.

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, versión 15.3 y posteriores (disponible en el modo `/std:c++17` y posteriores): una instrucción `switch` puede tener una cláusula `init-statement`, que termina con un punto y coma. Introduce e inicializa una variable cuyo ámbito está limitado al bloque de la instrucción `switch`:

C++

```

switch (Gadget gadget(args); auto s = gadget.get_status())
{
    case status::good:
        gadget.zip();
        break;
    case status::bad:
        throw BadGadget();
}

```

Un bloque interno de una instrucción `switch` puede contener definiciones con inicializadores siempre que sean *accesibles*, es decir, siempre que las rutas de ejecución posibles no las omitan. Los nombres proporcionados mediante estas declaraciones tienen ámbito local. Por ejemplo:

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;
}
}
```

Una instrucción `switch` puede estar anidada. Cuando está anidada, las etiquetas `case` o `default` se asocian con la instrucción `switch` más cercana que las contenga.

Comportamiento específico de Microsoft

Microsoft C++ no limita el número de valores `case` de una instrucción `switch`. El número solo está limitado por la memoria disponible.

Consulte también

[Instrucciones de selección](#)

[Palabras clave](#)

Instrucciones de iteración (C++)

Artículo • 26/09/2022 • Tiempo de lectura: 2 minutos

Las instrucciones de iteración producen instrucciones (o instrucciones compuestas) que se ejecutarán cero o más veces, según determinados criterios de la finalización de bucle. Cuando estas instrucciones son instrucciones compuestas, se ejecutan en orden, excepto cuando se encuentra la instrucción `break` o la instrucción `continue`.

C++ proporciona cuatro instrucciones de iteración: `while`, `do`, `for` y `range-based for`. Cada una de ellas se repite hasta que la expresión de finalización se evalúa como cero (false) o hasta que se fuerza la finalización del bucle con una instrucción `break`. En la tabla siguiente se resumen estas instrucciones y sus acciones; cada una se explica detalladamente en las secciones siguientes.

Instrucciones de iteración

| . | Se evalúa en | Inicialización | Incremento |
|--------------------------------------|---------------------|----------------|------------|
| <code>while</code> | Principio del bucle | No | No |
| <code>do</code> | Final del bucle | No | No |
| <code>for</code> | Principio del bucle | Sí | Sí |
| <code>for basado en intervalo</code> | Principio del bucle | Sí | Sí |

La parte de instrucción de una instrucción de iteración no puede ser una declaración. Sin embargo, puede ser una instrucción compuesta que contenga una declaración.

Consulte también

[Información general sobre las instrucciones de C++](#)

while (Instrucción) (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Ejecuta *statement* repetidamente hasta que *expression* se evalúa como cero.

Sintaxis

```
while ( expression )
    statement
```

Comentarios

La comprobación de *expression* tiene lugar antes de cada ejecución del bucle; por tanto, un bucle `while` se ejecuta cero o más veces. *expression* debe ser de tipo entero, un tipo de puntero o un tipo de clase con una conversión no ambigua a un tipo entero o de puntero.

Un bucle `while` también puede finalizar cuando se ejecuta `break`, `goto`, o `return` dentro del cuerpo de instrucción. Utilice `continue` para finalizar la iteración actual sin salir del bucle `while`. `continue` transfiere el control a la iteración siguiente del bucle `while`.

En el código siguiente se utiliza un bucle `while` para recortar los caracteres de subrayado finales de una cadena:

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));
}
```

La condición de finalización se evalúa al principio del bucle. Si no hay ningún carácter de subrayado final, el bucle nunca se ejecuta.

Consulte también

[Instrucciones de iteración](#)

[Palabras clave](#)

[do-while \(Instrucción\) \(C++\)](#)

[for \(Instrucción\) \(C++\)](#)

[Instrucción for basada en intervalo \(C++\)](#)

do-while (instrucción de C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Ejecuta un elemento *statement* repetidamente hasta que la condición de finalización (*la expresión*) se evalúa como cero.

Sintaxis

```
do
    statement
while ( expression ) ;
```

Comentarios

La prueba de la condición de finalización se realiza después de cada ejecución del bucle; por consiguiente, un bucle **do-while** se ejecuta una o más veces, dependiendo del valor de la expresión de finalización. La instrucción **do-while** también puede finalizar cuando se ejecuta una instrucción **break**, **goto** o **return** dentro del cuerpo de la instrucción.

expression debe tener un tipo aritmético o de puntero. La ejecución continúa de la siguiente manera:

1. Se ejecuta el cuerpo de instrucción.
2. A continuación, se evalúa *expression*. Si *expression* es false, la instrucción **do-while** finaliza y el control pasa a la siguiente instrucción del programa. Si *expression* es true (distinta de cero), el proceso se repite a partir del paso 1.

Ejemplo

En el siguiente ejemplo se muestra la instrucción **do-while**:

C++

```
// do_while_statement.cpp
#include <stdio.h>
int main()
{
    int i = 0;
    do
```

```
{  
    printf_s("\n%d",i++);  
} while (i < 3);  
}
```

Consulte también

[Instrucciones de iteración](#)

[Palabras clave](#)

[while \(Instrucción\) \(C++\)](#)

[for \(Instrucción\) \(C++\)](#)

[Instrucción for basada en intervalo \(C++\)](#)

Instrucción `for` (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 4 minutos

Ejecuta una instrucción repetidamente hasta que la condición es false. Para obtener información sobre la instrucción `for` basada en intervalos, consulte [Instrucción for \(C++\) basada en intervalos](#). Para obtener más información sobre la instrucción `for each` de C++/CLI, consulte [for each, in](#).

Sintaxis

```
for ( init-expression ; cond-expression ; Loop-expression )  
    statement
```

Comentarios

Use la instrucción `for` para construir bucles que deben ejecutarse un número especificado de veces.

La instrucción `for` consta de tres partes opcionales, como se muestra en la tabla siguiente.

Elementos del bucle `for`

| Nombre de la sintaxis | Cuándo se ejecuta | Descripción |
|------------------------------|---|---|
| <code>init-expression</code> | Antes que cualquier otro elemento de la instrucción <code>for</code> , <code>init-expression</code> solo se ejecuta una vez. El control pasa entonces a <code>cond-expression</code> . | Se suele utilizar para inicializar índices de bucle. Puede contener expresiones o declaraciones. |
| <code>cond-expression</code> | Antes de la ejecución de cada iteración de <code>statement</code> , incluida la primera iteración. <code>statement</code> solo se ejecuta si <code>cond-expression</code> se evalúa como true (distinto de cero). | Expresión que se evalúa como un tipo entero o un tipo de clase que tiene una conversión no ambigua a un tipo entero. Se utiliza normalmente para comprobar los criterios de finalización del bucle. |

| Nombre de la sintaxis | Cuándo se ejecuta | Descripción |
|-----------------------------|--|--|
| | <p><code>Loop-expression</code></p> <p>Al final de cada iteración de <code>statement</code>. Después de ejecutarse <code>Loop-expression</code>, se evalúa <code>cond-expression</code>.</p> | <p>Se utiliza normalmente para incrementar índices de bucle.</p> |

En los ejemplos siguientes se muestran distintas formas de usar la instrucción `for`.

C++

```
#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` y `Loop-expression` pueden contener varias instrucciones separadas por comas. Por ejemplo:

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 se puede aumentar o reducir, o modificar de otras maneras.

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
```

Un bucle `for` finaliza cuando se ejecuta una instrucción `break`, `return` o `goto` (en una instrucción con etiqueta fuera del bucle `for`) dentro de `statement`. Una instrucción `continue` en un bucle `for` solo finaliza la iteración actual.

Si se omite `cond-expression`, se considera como `true` y el bucle `for` no terminará sin `break`, `return` o `goto` en `statement`.

Aunque los tres campos de la instrucción `for` se suelen usar para la inicialización, comprobar la finalización y el incremento, no se limitan a estos usos. Por ejemplo, el código siguiente imprime los números de 0 a 4. En este caso, `statement` es la instrucción NULL:

C++

```
#include <iostream>  
using namespace std;  
  
int main()  
{  
    int i;  
    for( i = 0; i < 5; cout << i << '\n', i++){  
        ;  
    }  
}
```

Bucles `for` y el estándar de C++

El estándar de C++ indica que una variable declarada en un bucle `for` saldrá del ámbito cuando finalice el bucle `for`. Por ejemplo:

```
C++

for (int i = 0 ; i < 5 ; i++) {
    // do something
}
// i is now out of scope under /Za or /Zc:forScope
```

De forma predeterminada, en `/Ze`, una variable declarada en un bucle `for` permanece en el ámbito hasta que finaliza el ámbito de inclusión del bucle `for`.

`/Zc:forScope` habilita el comportamiento estándar de las variables declaradas en bucles `for` sin necesidad de especificar `/za`.

También es posible utilizar las diferencias de ámbito del bucle `for` para volver a declarar variables en `/ze` de la manera siguiente:

```
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++ ) {}
}
```

Esto imita mejor el comportamiento estándar de una variable declarada en un bucle `for`, que requiere que las variables declaradas en un bucle `for` salgan del ámbito cuando finalice el bucle. Cuando una variable se declara en un bucle `for`, el compilador la promueve internamente a una variable local en el ámbito de inclusión del bucle `for` incluso aunque ya exista una variable local con el mismo nombre. Se promueve incluso si ya hay una variable local con el mismo nombre.

Consulte también

[Instrucciones de iteración](#)

[Palabras clave](#)

Instrucción while (C++)

Instrucción do-while (C++)

Instrucción for (C++) basada en intervalos

Instrucción for basada en intervalo (C++)

Artículo • 26/09/2022 • Tiempo de lectura: 3 minutos

Ejecuta `statement` de forma repetida y secuencial para cada elemento de `expression`.

Sintaxis

```
for (for-range-declaration : expression)
    instrucción
```

Comentarios

Use la instrucción `for` basada en intervalo para construir los bucles que deben ejecutarse en un *intervalo*, que se define como cualquier elemento que se puede recorrer en iteración; por ejemplo, `std::vector` o cualquier otra secuencia de la biblioteca estándar de C++ cuyo intervalo esté definido por `begin()` y `end()`. El nombre que se declara en la parte `for-range-declaration` es local de la instrucción `for` y no se puede volver a declarar en `expression` o `statement`. Tenga en cuenta que es preferible usar la palabra clave `auto` en la parte `for-range-declaration` de la instrucción.

Novedades de Visual Studio 2017: los bucles `for` basados en intervalos ya no necesitan que `begin()` ni `end()` devuelvan objetos del mismo tipo. Esto permite que `end()` devuelva un objeto centinela como el que usan los intervalos tal como se define en la propuesta de intervalos V3. Para obtener más información, consulte [Generalización del bucle For basado en intervalo](#) y la [Biblioteca range-v3 en GitHub](#).

En este código se muestra cómo usar bucles `for` basados en intervalos para iterar por una matriz y un vector:

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;
}

```

Este es el resultado:

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
```

Un bucle `for` basado en intervalo finaliza cuando se ejecuta uno de estos elementos en `statement`: `break`, `return` o `goto` a una instrucción con etiqueta fuera del bucle `for` basado en intervalo. Una instrucción `continue` en un bucle `for` basado en intervalo solo finaliza la iteración actual.

Tenga en cuenta lo siguiente sobre la instrucción `for` basada en intervalo:

- Reconoce automáticamente las matrices.
- Reconoce los contenedores que tienen `.begin()` y `.end()`.
- Utiliza la búsqueda dependiente de argumentos `begin()` y `end()` para todo lo demás.

Consulte también

[auto](#)

[Instrucciones de iteración](#)

[Palabras clave](#)

[Instrucción while \(C++\)](#)

[Instrucción do-while \(C++\)](#)

[Instrucción for \(C++\)](#)

Instrucciones de salto (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Una instrucción de salto de C++ realiza una transferencia de control local inmediata.

Sintaxis

```
break;  
continue;  
return [expression];  
goto identifier;
```

Comentarios

Vea los temas siguientes para obtener una descripción de las instrucciones de salto de C++.

- [break \(Instrucción\)](#)
- [continue \(Instrucción\)](#)
- [return \(instrucción\)](#)
- [Instrucción goto](#)

Consulte también

[Información general sobre las instrucciones de C++](#)

break (Instrucción) (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

La instrucción `break` finaliza la ejecución del bucle o la instrucción condicional envolvente más próximo en el que aparece. El control pasa a la instrucción que hay a continuación del final de la instrucción, si hay alguna.

Sintaxis

```
break;
```

Comentarios

La instrucción `break` se utiliza con la instrucción condicional `switch` y con las instrucciones de bucle `do`, `for` y `while`.

En una instrucción `switch`, la instrucción `break` hace que el programa ejecute la siguiente instrucción que hay fuera de la instrucción `switch`. Sin una instrucción `break`, se ejecutan todas las instrucciones que hay desde la etiqueta `case` coincidente hasta el final de la instrucción `switch`, incluida la cláusula `default`.

En los bucles, la instrucción `break` finaliza la ejecución de la instrucción envolvente `do`, `for` o `while` más próxima. El control pasa a la instrucción que hay a continuación de la instrucción finalizada, si hay alguna.

Dentro de instrucciones anidadas, la instrucción `break` finaliza solo la instrucción `do`, `for`, `switch` o `while` que la envuelve inmediatamente. Puede utilizar una instrucción `return` o `goto` para transferir el control desde estructuras más anidadas.

Ejemplo

En el código siguiente se muestra cómo usar la instrucción `break` en un bucle `for`.

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

In each case:

1
2
3

En el código siguiente se muestra cómo usar `break` en un bucle `while` y un bucle `do`.

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

In each case:

0123

En el código siguiente se muestra cómo usar `break` en una instrucción switch. Debe usar `break` en todos los casos si desea tratar cada caso por separado; si no emplea `break`, la ejecución de código pasa al caso siguiente.

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";
    }
}
```

Consulte también

[Instrucciones de salto](#)

[Palabras clave](#)

[continue \(Instrucción\)](#)

continue (Instrucción) (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Fuerza la transferencia del control a la expresión de control del bucle contenido o `do`, `for` o `while` más pequeño.

Sintaxis

```
continue;
```

Comentarios

No se ejecuta ninguna de las instrucciones restantes de la iteración actual. La siguiente iteración del bucle se determina del modo siguiente:

- En un bucle `do` o `while`, la siguiente iteración se inicia reevaluando la expresión de control de la instrucción `do` o `while`.
- En un bucle `for` (que use la sintaxis `for(<init-expr> ; <cond-expr> ; <loop-expr>)`), se ejecuta la cláusula `<loop-expr>`. A continuación, se evalúa de nuevo la cláusula `<cond-expr>` y, en función del resultado, el bucle finaliza o se produce otra iteración.

En el ejemplo siguiente se muestra cómo se puede usar la instrucción `continue` para omitir secciones de código e iniciar la siguiente iteración de un bucle.

Ejemplo

C++

```
// continue_statement.cpp
#include <stdio.h>
int main()
{
    int i = 0;
    do
    {
        i++;
        printf_s("before the continue\n");
        continue;
    }
}
```

```
    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
```

Consulte también

[Instrucciones de salto](#)

[Palabras clave](#)

return (Instrucción) (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Finaliza la ejecución de una función y devuelve el control a la función de llamada (o al sistema operativo si se transfiere el control de la función `main`). La ejecución se reanuda en la función de llamada, en el punto que sigue inmediatamente a la llamada.

Sintaxis

```
return [expression];
```

Comentarios

La cláusula `expression`, si está presente, se convierte al tipo especificado en la declaración de función, como si se realizara una inicialización. La conversión del tipo de la expresión al tipo `return` de la función puede crear objetos temporales. Para obtener más información sobre cómo y cuándo se crean los elementos temporales, consulte [Objetos temporales](#).

El valor de la cláusula `expression` se devuelve a la función de llamada. Si se omite la expresión, el valor devuelto de la función es indefinido. Los constructores y destructores, así como las funciones de tipo `void`, no pueden especificar una expresión en la instrucción `return`. Las funciones de todos los demás tipos deben especificar una expresión en la instrucción `return`.

Cuando el flujo de control sale del bloque que incluye la definición de función, el resultado es el mismo que si se hubiera ejecutado una instrucción `return` sin una expresión. Esto no es válido para las funciones que se declaran como si devolvieran un valor.

Una función puede tener cualquier número de instrucciones `return`.

En el ejemplo siguiente se usa una expresión con una instrucción `return` para obtener el mayor de dos enteros.

Ejemplo

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 ));
}
```

Consulte también

[Instrucciones de salto](#)

[Palabras clave](#)

goto (Instrucción) (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

La instrucción `goto` transfiere incondicionalmente el control a la instrucción etiquetada por el identificador especificado.

Sintaxis

```
goto identifier;
```

Comentarios

La instrucción con etiqueta designada por `identifier` debe estar en la función actual. Todos los nombres de `identifier` son miembros de un espacio de nombres interno y, por tanto, no interfieren con otros identificadores.

Una etiqueta de instrucción solo es significativa para una instrucción `goto`; de lo contrario, se omiten las etiquetas de instrucciones. Las etiquetas no se pueden volver a declarar.

No se permite que una instrucción `goto` transfiera el control a una ubicación que omita la inicialización de cualquier variable que esté en el ámbito de esa ubicación. En el ejemplo siguiente se produce el error C2362:

C++

```
int goto_fn(bool b)
{
    if (!b)
    {
        goto exit; // C2362
    }
    else
    { /*...*/ }

    int error_code = 42;

exit:
    return error_code;
}
```

Una buena recomendación de programación es utilizar las instrucciones `break`, `continue` y `return` en lugar de la instrucción `goto` siempre que sea posible. Sin embargo, puesto que la instrucción `break` sale de un solo nivel de un bucle, puede que tenga que utilizar una instrucción `goto` para salir de un bucle anidado profundamente.

Para más información sobre las etiquetas y la instrucción `goto`, consulte [Instrucciones con etiquetas](#).

Ejemplo

En este ejemplo, una instrucción `goto` transfiere el control al punto con la etiqueta `stop` cuando `i` es igual a 3.

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
```

Consulte también

[Instrucciones de salto](#)

[Palabras clave](#)

Transferencias del control

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Puede utilizar la instrucción `goto` o una etiqueta `case` en una instrucción `switch` para especificar un programa que se bifurque más allá de un inicializador. Este código no es válido a menos que la declaración que contenga el inicializador esté en un bloque dentro del bloque en el que aparezca la instrucción de salto.

En el ejemplo siguiente se muestra un bucle que declara e inicializa los objetos `total`, `ch` y `i`. Hay también una instrucción `goto` errónea que transfiere el control más allá de un inicializador.

```
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;  
    }  
}
```

En el ejemplo anterior, la instrucción `goto` intenta transferir el control más allá de la inicialización de `i`. Sin embargo, si se declarara `i` pero no se inicializara, la transferencia sería válida.

Los objetos `total` y `ch`, declarados en el bloque que sirve como *statement* de la instrucción `while`, se destruyen cuando se usa la instrucción `break` para salir del bloque.

Espacios de nombres (C++)

Artículo • 26/09/2022 • Tiempo de lectura: 8 minutos

Un espacio de nombres es una región declarativa que proporciona un ámbito a los identificadores (nombres de tipos, funciones, variables, etc.) de su interior. Los espacios de nombres se utilizan para organizar el código en grupos lógicos y para evitar conflictos de nombres que pueden producirse, especialmente cuando la base de código incluye varias bibliotecas. Todos los identificadores del ámbito del espacio de nombres son visibles entre sí sin calificación. Los identificadores que están fuera del espacio de nombres pueden tener acceso a los miembros si usan el nombre completo de cada identificador (por ejemplo `std::vector<std::string> vec;`), o bien mediante una declaración "using" para un identificador único (`using std::string`) o una directiva `using` para todos los identificadores del espacio de nombres (`using namespace std;`). El código de los archivos de encabezado debe utilizar siempre el nombre completo del espacio de nombres.

En el ejemplo siguiente se muestra una declaración de espacio de nombres y tres formas de que el código que está fuera del espacio de nombres obtenga acceso a sus miembros.

C++

```
namespace ContosoData
{
    class ObjectManager
    {
        public:
            void DoSomething() {}
    };
    void Func(ObjectManager) {}
}
```

Use el nombre completo:

C++

```
ContosoData::ObjectManager mgr;
mgr.DoSomething();
ContosoData::Func(mgr);
```

Use una declaración using para poner un identificador en el ámbito:

C++

```
using ContosoData::ObjectManager;  
ObjectManager mgr;  
mgr.DoSomething();
```

Use una directiva `using` para poner todo el espacio de nombres en el ámbito:

C++

```
using namespace ContosoData;  
  
ObjectManager mgr;  
mgr.DoSomething();  
Func(mgr);
```

Directivas using

La directiva `using` permite usar todos los nombres en un `namespace` sin emplear *namespace-name* como calificador explícito. Use una directiva `using` en un archivo de implementación (p. ej. `.*.cpp`) si está utilizando varios identificadores diferentes en un espacio de nombres. Si solo está usando uno o dos identificadores, considere la posibilidad de usar una declaración `using` para poner solo esos identificadores en el ámbito, y no todos los identificadores del espacio de nombres. Si una variable local tiene el mismo nombre que una variable de espacio de nombres, se oculta la variable de espacio de nombres. Es un error tener una variable de espacio de nombres con el mismo nombre que una variable global.

ⓘ Nota

Una directiva `using` puede colocarse en la parte superior del archivo `.cpp` (en el ámbito del archivo), o dentro de una definición de clase o función.

En general, evite colocar directivas `using` en un archivo de encabezado (`*.h`) porque cualquier archivo que incluya ese encabezado pondrá todo en el espacio de nombres en el ámbito, lo que puede ocasionar problemas de ocultación de nombres y colisión de nombres que son muy difíciles de depurar. Utilice siempre nombres completos en los archivos de encabezado. Si esos nombres acaban siendo demasiado largos, puede utilizar un alias de espacio de nombres para acortarlos. (Vea a continuación).

Declarar espacios de nombres y miembros de espacio de nombres

Normalmente, los espacios de nombres se declaran en un archivo de encabezado. Si las implementaciones de sus funciones están en un archivo independiente, complete los nombres de función, como en este ejemplo.

C++

```
//contosoData.h
#pragma once
namespace ContosoDataServer
{
    void Foo();
    int Bar();
}
```

Las implementaciones de funciones en contosodata.cpp deben usar el nombre completo, incluso si coloca una directiva `using` en la parte superior del archivo:

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;}
```

Se puede declarar un espacio de nombres en varios bloques de un solo archivo y en varios archivos. El compilador une las partes durante el preprocessamiento y el espacio de nombres resultante contiene todos los miembros declarados en todas las partes. Un ejemplo de esto es el espacio de nombres `std`, que se declara en cada uno de los archivos de encabezado de la biblioteca estándar.

Los miembros de un espacio de nombres con nombre pueden definirse fuera del espacio de nombres en el que se declaran por calificación explícita del nombre que se define. Sin embargo, la definición debe aparecer después del punto de la declaración de un espacio de nombres que incluye el espacio de nombres de la declaración. Por ejemplo:

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();
}
```

Este error puede producirse cuando los miembros del espacio de nombres se declaran en varios archivos de encabezado y estos encabezados no se han incluido en el orden correcto.

El espacio de nombres global

Si un identificador no se declara en un espacio de nombres explícito, forma parte del espacio de nombres global implícito. En general, intente evitar las declaraciones en el ámbito global siempre que pueda, salvo con la [función main](#) de punto de entrada, que necesita estar en el espacio de nombres global. Para calificar explícitamente un identificador global, utilice el operador de resolución de ámbito sin nombre, como en `::SomeFunction(x);`. Así, el identificador se diferenciará de cualquier elemento que tenga el mismo nombre en otro espacio de nombres, y también facilitará que otras personas entiendan el código.

El espacio de nombres std

Todas las funciones y tipos de la biblioteca estándar de C++ se declaran en el espacio de nombres `std` o en espacios de nombres anidados dentro de `std`.

Espacios de nombres anidados

Los espacios de nombres pueden estar anidados. Un espacio de nombres anidado normal tiene acceso incompleto a los miembros de su elemento primario, pero los miembros primarios no tienen acceso incompleto al espacio de nombres anidado (a menos que se declare como alineado), como se muestra en el siguiente ejemplo:

C++

```
namespace ContosoDataServer
{
    void Foo();

    namespace Details
    {
        int CountImpl;
        void Ban() { return Foo(); }
    }

    int Bar(){...};
    int Baz(int i) { return Details::CountImpl; }
}
```

Los espacios de nombres anidados normales pueden utilizarse para encapsular los detalles de implementación internos que no forman parte de la interfaz pública del espacio de nombres primario.

Espacios de nombres alineados (C++ 11)

A diferencia de un espacio de nombres anidado normal, los miembros de un espacio de nombres alineado se tratan como miembros del espacio de nombres primario. Esta característica permite la búsqueda dependiente de argumentos en funciones sobrecargadas para trabajar con funciones que tienen sobrecargas en un elemento primario y un espacio de nombres anidado alineado. También permite declarar una especialización en un espacio de nombres primario para una plantilla que se declara en el espacio de nombres alineado. En el ejemplo siguiente se muestra cómo el código externo enlaza de forma predeterminada con el espacio de nombres alineado:

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;
}

```

El siguiente ejemplo muestra cómo se puede declarar una especialización en un elemento primario de una plantilla que se declara en un espacio de nombres alineado:

C++

```

namespace Parent
{
    inline namespace new_ns
    {
        template <typename T>
        struct C
        {
            T member;
        };
        template<>
        class C<int> {};
    }
}

```

Puede usar espacios de nombres alineados como mecanismo de control de versiones para administrar los cambios en la interfaz pública de una biblioteca. Por ejemplo, puede crear un espacio de nombres primario único y encapsular cada versión de la interfaz en su propio espacio de nombres anidado dentro del elemento primario. El espacio de nombres que contiene la versión más reciente o preferida se califica como alineado y, por tanto, se expone como si fuera un miembro directo del espacio de nombres primario. El código del cliente que invoca la clase Parent::Class se enlazará automáticamente al nuevo código. Los clientes que prefieren usar la versión anterior siguen teniendo acceso a ella mediante la ruta de acceso completa al espacio de nombres anidado que contiene ese código.

La palabra clave `inline` se debe aplicar a la primera declaración del espacio de nombres en una unidad de compilación.

El ejemplo siguiente muestra dos versiones de una interfaz, cada una en un espacio de nombres anidado. El espacio de nombres `v_20` tiene algunas modificaciones en la interfaz `v_10` y se marca como alineado. El código de cliente que utiliza la nueva biblioteca y llama a `Contoso::Funcs::Add` invocará la versión `v_20`. El código que intente llamar a `Contoso::Funcs::Divide` obtendrá ahora un error de tiempo de compilación. Si realmente necesita esa función, puede obtener acceso a la versión `v_10` llamando explícitamente a `Contoso::v_10::Funcs::Divide`.

```
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);  
        };  
    }  
}
```

Alias de espacios de nombres

Los nombres de los espacios de nombres deben ser únicos, lo que significa que a menudo no pueden ser demasiado cortos. Si la longitud de un nombre hace que el código sea difícil de leer, o resulta tedioso escribirlo en un archivo de encabezado

donde no se pueden usar directivas using, puede hacer un alias del espacio de nombres que actúe como una abreviatura del nombre real. Por ejemplo:

C++

```
namespace a_very_long_namespace_name { class Foo {}; }
namespace AVLNN = a_very_long_namespace_name;
void Bar(AVLNN::Foo foo){ }
```

espacios de nombres anónimos o sin nombre

Puede crear un espacio de nombres explícito, pero sin asignarle un nombre:

C++

```
namespace
{
    int MyFunc(){}
}
```

Esto se denomina espacio de nombres sin nombre o anónimo, y resulta útil cuando desea que las declaraciones de variable no sean visibles para el código de otros archivos (p. ej. otorgarles vinculación interna) sin tener que crear un espacio de nombres designado. Todo el código del mismo archivo puede ver los identificadores en un espacio de nombres sin nombre, pero los identificadores, junto con el espacio de nombres, no son visibles fuera de ese archivo, o más concretamente fuera de la unidad de traducción.

Consulte también

[Declaraciones y definiciones](#)

Enumeraciones [C++]

Artículo • 03/03/2023 • Tiempo de lectura: 6 minutos

Una enumeración es un tipo definido por el usuario que consta de un conjunto de constantes enteras con nombre conocidas como *enumeradores*.

ⓘ Nota

Este artículo trata el tipo `enum` del lenguaje C++ estándar de ISO y el tipo de ámbito (o fuertemente tipado) `enum class`, que se introdujo en C++11. Para obtener información acerca de los tipos `public enum class` o `private enum class` en C++/CLI y C++/CX, vea [enum class \(C++/CLI y C++/CX\)](#).

Sintaxis

`enum-name`:

`identifier`

`enum-specifier`:

`enum-head { enumerator-list opt }`
`enum-head { enumerator-list , }`

`enum-head`:

`enum-key attribute-specifier-seq opt enum-head-name opt enum-base opt`

`enum-head-name`:

`nested-name-specifier opt identifier`

`opaque-enum-declaration`:

`enum-key attribute-specifier-seq opt enum-head-name enum-base opt ;`

`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

Uso

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
```

Parámetros

identifier

Nombre del tipo dado a la enumeración.

type

El tipo subyacente de los enumeradores; cada enumerador tiene el mismo tipo subyacente. Puede ser cualquier tipo entero.

enum-list

Una lista delimitada por comas de los enumeradores en la enumeración. Cada enumerador o nombre de variable en el ámbito debe ser único. Sin embargo, los valores pueden estar duplicados. En una enumeración sin ámbito, el ámbito es el ámbito adyacente; en una enumeración con ámbito, el ámbito es el mismo elemento *enum-list*. En una enumeración con ámbito, la lista puede estar vacía, que en efecto define un nuevo tipo entero.

```
class
```

Al usar esta palabra clave en la declaración, especifica que la enumeración se incluye en el ámbito, por lo que es necesario proporcionar *identifier*. También puede usar la palabra clave **struct** en lugar de **class**, ya que son equivalentes semánticas en este contexto.

Ámbito del enumerador

Una enumeración proporciona contexto para describir un intervalo de valores que se representan como constantes con nombre. Estas constantes con nombre también se denominan *enumeradores*. En los tipos **enum** originales de C y C++, los enumeradores incompletos están visibles en el ámbito en el que se declara **enum**. En enumeraciones de ámbito, el nombre del enumerador debe calificarse con el nombre de tipo **enum**. El ejemplo siguiente muestra esta diferencia básica entre las dos clases de enumeraciones:

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
        { /*...*/
        }
    }
}
```

A cada nombre de la enumeración se le asigna un valor entero que corresponde al lugar que ocupa en el orden de los valores de la enumeración. De forma predeterminada, al primer valor se asigna 0, al siguiente se asigna 1 y así sucesivamente, pero puede establecer explícitamente el valor de un enumerador, como se muestra aquí:

C++

```
enum Suit { Diamonds = 1, Hearts, Clubs, Spades };
```

El enumerador `Diamonds` tiene asignado el valor `1`. Los enumeradores subsiguientes, si no se les asigna un valor explícito, reciben el valor del enumerador anterior más uno. En el ejemplo anterior, `Hearts` tendría el valor `2`, `Clubs` tendría `3`, etc.

Cada enumerador se trata como una constante y debe tener un nombre único dentro del ámbito, donde `enum` está definido (para las enumeraciones sin ámbito) o en el propio elemento `enum` (para las enumeraciones de ámbito). Los valores especificados en los nombres no tienen que ser únicos. Por ejemplo, considere esta declaración de una enumeración `Suit` sin ámbito:

C++

```
enum Suit { Diamonds = 5, Hearts, Clubs = 4, Spades };
```

Los valores de `Diamonds`, `Hearts`, `Clubs` y `Spades` son `5`, `6`, `4` y `5`, respectivamente.

Observe que `5` se usa más de una vez; esto se permite incluso aunque pueda no ser intencionado. Estas reglas son las mismas para las enumeraciones de ámbito.

Reglas de conversión

Las constantes de enumeración sin ámbito se pueden convertir implícitamente a `int`, pero `int` nunca es implícitamente convertible a un valor `enum`. El ejemplo siguiente muestra lo que ocurre si intenta asignar a `hand` un valor que no sea `Suit`:

C++

```
int account_num = 135692;
Suit hand;
hand = account_num; // error C2440: '=' : cannot convert from 'int' to
                     'Suit'
```

Se requiere una conversión para convertir un `int` a un enumerador con ámbito o sin ámbito. Pero puede promover un enumerador sin ámbito a un valor entero sin una conversión.

C++

```
int account_num = Hearts; //OK if Hearts is in a unscooped enum
```

Utilizar conversiones implícitas de esta manera puede provocar efectos secundarios imprevistos. Para ayudar a eliminar los errores de programación asociados a las enumeraciones sin ámbito, los valores de ámbito de enumeración están fuertemente tipados. Los enumeradores con ámbito deben calificarse por el nombre de tipo de enumeración (identificador) y no pueden convertirse implícitamente, como se muestra en el ejemplo siguiente:

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
    }
}
```

Observe que la línea `hand = account_num;` aún produce el error que se produce con enumeraciones sin ámbito, como se muestra anteriormente. Esto se permite con una conversión explícita. Sin embargo, con las enumeraciones con ámbito, la conversión que se ha intentado en la siguiente instrucción, `account_num = Suit::Hearts;`, ya no se permite sin una conversión explícita.

Enumeraciones sin enumeradores

Visual Studio 2017 versión 15.3 y posteriores (disponible con [/std:c++17](#) y versiones posteriores): al definir una enumeración (normal o con ámbito) con un tipo subyacente explícito y sin enumeradores, puede introducir un nuevo tipo entero que no tenga ninguna conversión implícita a ningún otro tipo. Mediante el uso de este tipo en lugar de su tipo subyacente integrado, puede eliminar la posibilidad de errores sutiles causados por conversiones implícitas involuntarias.

C++

```
enum class byte : unsigned char { };
```

El nuevo tipo es una copia exacta del tipo subyacente y, por tanto, tiene la misma convención de llamada, lo que significa que se puede usar en las ABI sin ninguna penalización de rendimiento. No se requiere ninguna conversión cuando se inicializan variables del tipo mediante la inicialización de lista directa. En el ejemplo siguiente se muestra cómo inicializar enumeraciones sin enumeradores en varios contextos:

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;
}
```

Consulte también

[Declaraciones de enumeración de C](#)

[Palabras clave](#)

union

Artículo • 03/03/2023 • Tiempo de lectura: 8 minutos

ⓘ Nota

En C++17 y las versiones posteriores, `std::variant` es una alternativa segura para tipos union.

`union` es un tipo definido por el usuario en el que todos los miembros comparten la misma ubicación de memoria. Esta definición significa que, en un momento dado, una union no puede contener más de un objeto de su lista de miembros. También significa que, independientemente de cuántos miembros tiene una union, en todo momento usa únicamente la memoria suficiente para almacenar al miembro más grande.

Una union pueden ser útiles para conservar memoria cuando se tienen muchos objetos y memoria limitada. Sin embargo, un union requiere un cuidado adicional para usarse correctamente. Usted es responsable de asegurarse de que siempre tenga acceso al mismo miembro que asignó. Si los tipos de miembro tienen un struct no trivial, debe escribir código adicional para struct y destruir ese miembro explícitamente. Antes de usar una union, considere si el problema que intenta resolver se puede expresar mejor usando una class base y tipos de class derivadas.

Sintaxis

```
union tag opt { member-list };
```

Parámetros

`tag`

Nombre del tipo asignado a la union.

`member-list`

Miembros que puede contener la union.

Declarar una union

Comience la declaración de una union mediante el uso de la palabra clave `union` y agregue la lista de miembros entre llaves:

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
}
```

Usar una union

En el ejemplo anterior, el código que acceda a la union debe saber qué miembro mantiene los datos. La solución más común a este problema se denomina una *union discriminada*. Incluye la union en un elemento struct, e incluye un miembro enum que indica el tipo de miembro almacenado actualmente en union. En el ejemplo siguiente se muestra el patrón básico:

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;  
}
```

En el ejemplo anterior, la union en el elemento `Input` struct no tiene nombre, por lo que se denomina *anónimounion*. Se puede acceder a sus miembros directamente como si fueran miembros de struct. Para obtener más información sobre cómo usar un anónimo union, consulte la sección [Anónimo union](#).

En el ejemplo anterior se muestra un problema que también se podría resolver mediante el uso de los tipos class que derivan de una base class común. Puede bifurcar el código en función del tipo en tiempo de ejecución de cada objeto del contenedor. El código puede ser más fácil de mantener y reconocer, pero también más lento que el uso de union. Además, con un union, se pueden almacenar tipos no relacionados. Una union permite cambiar dinámicamente el tipo del valor almacenado sin cambiar el tipo de la propia variable union. Por ejemplo, se podría crear una matriz heterogénea de `MyUnionType`, cuyos elementos almacenan diferentes valores de tipos diferentes.

Es fácil usar incorrectamente el `Input` struct en el ejemplo. El usuario es quien debe usar el discriminador correctamente para acceder al miembro que contiene los datos. Para evitar el uso indebido, haga que la union privada `private` y proporcione funciones de acceso especiales, como se muestra en el ejemplo siguiente.

Sin restricción union (C++11)

En C++03 y versiones anteriores, una union puede contener miembros de datos no static que tienen un tipo class siempre que el tipo no tenga constructores, destructores ni operadores de asignación proporcionados por el usuario. En C++11, se quitaron estas restricciones. Si incluye miembros de este tipo en la union, el compilador marcará automáticamente las funciones miembro especiales que no proporcionase el usuario como `deleted`. Si la union es una union anónima dentro de una class o una struct, entonces cualquier función miembro especial de la class o la struct que no sea proporcionada por el usuario se marca como `deleted`. En el ejemplo siguiente se muestra cómo manejar ese caso. Uno de los miembros de la union tiene un miembro que requiere este tratamiento especial:

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;
}

```

Una union no puede almacenar una referencia. Tampoco union admite la herencia. Esto significa que no se puede usar union como base class o heredar de otra class, o bien tener funciones virtuales.

Inicializa un union

Puede declarar e inicializar una union en la misma instrucción mediante la asignación de una expresión entre llaves. La expresión se evalúa y se asigna al primer campo de la union.

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;
}

```

```
}
```

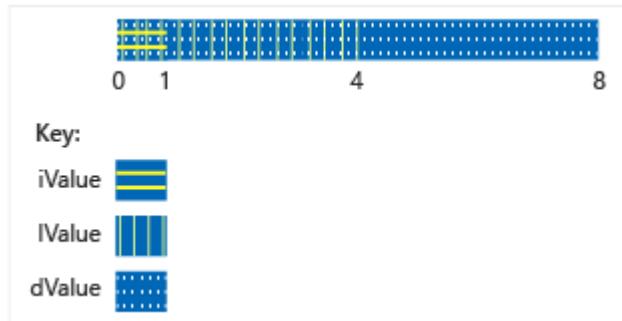
```
/* Output:
```

```
10
```

```
3.141600
```

```
*/
```

La `NumericType` union se organiza en memoria (conceptualmente), como se muestra en la ilustración siguiente.



Almacenamiento de datos en una `NumericType` union

union anónima

Una union anónima es una declarada sin un `class-name` o `declarator-list`.

```
union { member-list }
```

Los nombres declarados en una union anónima se usan directamente, como las variables no miembro. Implica que los nombres declarados en una union anónima deben ser únicos en el ámbito circundante.

Una union anónima está sujeta a estas restricciones adicionales:

- Si se declaran en el ámbito de archivo o espacio de nombres, se debe declarar como `static`.
- Solo puede tener miembros `public`; el tener miembros `private` y `protected` en una union anónima genera errores.
- No puede tener funciones miembro.

Consulte también

[Clases y structs](#)

[Palabras clave](#)

class

struct

Funciones (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 14 minutos

Una función es un bloque de código que realiza alguna operación. Una función puede definir opcionalmente parámetros de entrada que permiten a los llamadores pasar argumentos a la función. Una función también puede devolver un valor como salida. Las funciones son útiles para encapsular las operaciones comunes en un solo bloque reutilizable, idealmente con un nombre que describa claramente lo que hace la función. La siguiente función acepta dos enteros de un autor de llamada y devuelve su suma; *a* y *b* son *parámetros* de tipo `int`.

C++

```
int sum(int a, int b)
{
    return a + b;
}
```

La función puede ser invocada, o *llamada*, desde cualquier lugar del programa. Los valores que se pasan a la función son los *argumentos*, cuyos tipos deben ser compatibles con los tipos de los parámetros en la definición de la función.

C++

```
int main()
{
    int i = sum(10, 32);
    int j = sum(i, 66);
    cout << "The value of j is" << j << endl; // 108
}
```

No hay ningún límite práctico para la longitud de la función, pero el buen diseño tiene como objetivo las funciones que realizan una sola tarea bien definida. Los algoritmos complejos deben dividirse en funciones más sencillas y fáciles de comprender siempre que sea posible.

Las funciones definidas en el ámbito de clase se denominan funciones miembro. En C++, a diferencia de otros lenguajes, una función también pueden definirse en el ámbito de espacio de nombres (incluido el espacio de nombres global implícito). Estas funciones se denominan *funciones libres* o *funciones no miembro*; se usan ampliamente en la biblioteca estándar.

Las funciones pueden ser *sobre cargadas*, lo que significa que diferentes versiones de una función pueden compartir el mismo nombre si difieren por el número y/o tipo de parámetros formales. Para obtener más información, consulte [Sobrecarga de funciones](#).

Elementos de una declaración de función

Una *declaración* de función mínima consta del tipo de valor devuelto, el nombre de la función y la lista de parámetros (que pueden estar vacíos), junto con palabras clave opcionales que proporcionan más instrucciones al compilador. El siguiente ejemplo es una declaración de función:

C++

```
int sum(int a, int b);
```

Una definición de función consiste en una declaración, más el *cuerpo*, que es todo el código entre las llaves:

C++

```
int sum(int a, int b)
{
    return a + b;
}
```

Una declaración de función seguida de un punto y coma puede aparecer en varios lugares de un programa. Debe aparecer antes de cualquier llamada a esa función en cada unidad de traducción. La definición de función debe aparecer solo una vez en el programa, según la regla de una definición (ODR).

Los elementos necesarios de una declaración de función son los siguientes:

1. El tipo de retorno, que especifica el tipo del valor que devuelve la función, o `void` si no se devuelve ningún valor. En C++11, `auto` es un tipo de retorno válido que indica al compilador que infiera el tipo de la sentencia de retorno. En C++14, `decltype(auto)` también está permitido. Para obtener más información, consulte más adelante Dedución de tipos en tipos de valor devueltos.
2. El nombre de la función, que debe comenzar con una letra o un carácter de subrayado y no puede contener espacios. En general, los caracteres de subrayado iniciales en los nombres de función de la biblioteca estándar indican funciones miembro privadas o funciones que no son miembro que no están diseñadas para su uso por parte del código.

3. La lista de parámetros, que es un conjunto delimitado por llaves y separado por comas de cero o más parámetros que especifican el tipo y, opcionalmente, un nombre local mediante el cual se puede acceder a los valores de dentro del cuerpo de la función.

Los elementos opcionales de una declaración de función son los siguientes:

1. **constexpr**, que indica que el valor devuelto de la función es un valor constante que se puede calcular en tiempo de compilación.

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. Su especificación de vinculación, **extern** o **static**.

C++

```
//Declare printf with C linkage.
extern "C" int printf( const char *fmt, ... );
```

Para obtener más información, consulte [Unidades de traducción y vinculación](#).

3. **inline**, que indica al compilador que reemplace todas las llamadas a la función con el propio código de la función. La inserción en línea puede mejorar el rendimiento en escenarios donde una función se ejecuta rápidamente y se invoca varias veces en una sección del código crítica para el rendimiento.

C++

```
inline double Account::GetBalance()
{
    return balance;
}
```

Para obtener más información, consulte [Funciones en línea](#).

4. Una **noexcept** expresión que especifica si la función puede o no lanzar una excepción. En el ejemplo siguiente, la función no produce una excepción si la

`is_pod` expresión se evalúa como `true`.

C++

```
#include <type_traits>

template <typename T>
T copy_object(T& obj) noexcept(std::is_pod<T>) {...}
```

Para más información, consulte [noexcept](#).

5. (solo funciones miembro) Los calificadores `cv`, que especifican si la función es `const` o `volatile`.

6. (solo funciones de los miembros) `virtual`, `override`, o `final`. `virtual` especifica que una función se puede reemplazar en una clase derivada. `override` significa que una función de una clase derivada reemplaza una función virtual. `final` significa que una función no se puede invalidar en ninguna clase derivada adicional. Para más información, consulte [Funciones virtuales](#).

7. (solo funciones miembro) `static` aplicado a una función miembro significa que la función no está asociada a ninguna instancia de objeto de la clase .

8. (solo funciones miembro no estáticas) El calificador de referencia, que especifica al compilador qué sobrecarga de una función, debe elegir cuando el parámetro implícito del objeto (`*this`) es una referencia rvalue frente a una referencia lvalue. Para obtener más información, consulte [Sobrecarga de funciones](#).

Definiciones de función

Una *definición de función* consiste en la declaración y el cuerpo de la función, encerrado entre llaves, que contiene declaraciones de variables, sentencias y expresiones. El siguiente ejemplo muestra una definición de función completa:

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;  
}
```

Las variables declaradas dentro del cuerpo se denominan variables locales. Se salen del ámbito cuando finaliza la función; por lo tanto, una función nunca debe devolver una referencia a una variable local.

C++

```
 MyClass& boom(int i, std::string s)  
{  
    int value {i};  
    MyClass mc;  
    mc.Initialize(i,s);  
    return mc;  
}
```

funciones const y constexpr

Puede declarar una función miembro como `const` para especificar que la función no puede cambiar los valores de los miembros de datos de la clase . Al declarar una función miembro como `const`, se ayuda al compilador a imponer la *const-corrección*. Si alguien intenta por error modificar el objeto utilizando una función declarada como `const`, se produce un error del compilador. Para obtener más información, consulte [const](#).

Declarar una función como `constexpr` cuando el valor que produce puede ser determinado en tiempo de compilación. Una función `constexpr` generalmente se ejecuta más rápido que una función regular. Para obtener más información, vea [constexpr](#).

Plantillas de función

Una plantilla de función es parecida a una plantilla de clase; genera funciones concretas que se basan en los argumentos de plantilla. En muchos casos, la plantilla es capaz de inferir los argumentos de tipo, por lo que no es necesario especificarlos de forma explícita.

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
```

Para obtener más información, consulte [Plantillas de funciones](#)

Parámetros de función y argumentos

Una función tiene una lista de parámetros separados por comas de cero o más tipos, cada uno de los cuales tiene un nombre mediante el cual se puede acceder a ellos dentro del cuerpo de la función. Una plantilla de función puede especificar más parámetros de tipo o valor. El llamador pasa argumentos, que son valores concretos cuyos tipos son compatibles con la lista de parámetros.

De forma predeterminada, los argumentos se pasan a la función por valor, lo que significa que la función recibe una copia del objeto que se pasa. En el caso de objetos grandes, la realización de una copia puede ser costosa y no siempre es necesaria. Para hacer que los argumentos se pasen por referencia (concretamente por referencia lvalue), agregue un calificador de referencia al parámetro:

C++

```
void DoSomething(std::string& input){...}
```

Cuando una función modifica un argumento que se pasa por referencia, modifica el objeto original, no una copia local. Para evitar que una función modifique un argumento de este tipo, califique el parámetro como const&:

C++

```
void DoSomething(const std::string& input){...}
```

C++ 11: Para manejar explícitamente los argumentos que se pasan por una referencia rvalue o lvalue, utilice un doble comercial en el parámetro para indicar una referencia universal:

C++

```
void DoSomething(const std::string&& input){...}
```

Una función declarada con la palabra clave única `void` en la lista de declaración de parámetros no toma argumentos, siempre que la palabra clave `void` sea el primer y único miembro de la lista de declaración de argumentos. Los argumentos del tipo `void` en otra parte de la lista producen errores. Por ejemplo:

C++

```
// OK same as GetTickCount()
long GetTickCount( void );
```

Aunque no es válido especificar un `void` argumento excepto como se describe aquí, los tipos derivados del tipo `void` (como punteros a `void` y matrices de `void`) pueden aparecer en cualquier lugar de la lista de declaraciones de argumentos.

Argumentos predeterminados

Es posible asignar un argumento predeterminado al último parámetro o parámetros de una firma de función, lo que significa que el llamador puede omitir el argumento cuando se llama a la función, a menos que desee especificar otro valor.

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)
{...}
```

Para obtener más información, consulte [Argumentos predeterminados](#).

Tipos de valor devuelto de función

Una función no puede devolver otra función, o una matriz incorporada; sin embargo, puede devolver punteros a estos tipos, o una *lambda*, que produce un objeto de

función. Excepto en estos casos, una función puede devolver un valor de cualquier tipo que esté en el ámbito, o puede no devolver ningún valor, en cuyo caso el tipo de retorno es `void`.

Tipos de valor devueltos finales

Un tipo de valor devuelto "normal" se encuentra en el lado izquierdo de la firma de función. Un *tipo de valor devuelto final* se encuentra en el lado derecho de la firma y va precedido por el `->` operador. Los tipos de valor devueltos finales son especialmente útiles en plantillas de función cuando el tipo del valor devuelto depende de los parámetros de plantilla.

C++

```
template<typename Lhs, typename Rhs>
auto Add(const Lhs& lhs, const Rhs& rhs) -> decltype(lhs + rhs)
{
    return lhs + rhs;
}
```

Cuando `auto` se usa junto con un tipo de valor devuelto final, sirve como marcador de posición para cualquier elemento que genere la expresión `decltype` y no realiza la deducción de tipos.

Variables locales de función

Una variable que se declara dentro del cuerpo de una función se llama *variable local* o simplemente *local*. Las variables locales no estáticas solo son visibles dentro del cuerpo de la función y, si se declaran en la pila, salen del ámbito cuando se cierra la función.

Cuando se construye una variable local y se devuelve por valor, el compilador suele realizar la *optimización del valor de retorno con nombre* para evitar operaciones de copia innecesarias. Si una variable local se devuelve por referencia, el compilador emitirá una advertencia, ya que cualquier intento por parte del llamador de usar esa referencia se producirá después de la destrucción de la variable local.

En C++, una variable local se puede declarar como estática. La variable solo es visible dentro del cuerpo de la función, pero existe una copia única de la variable para todas las instancias de la función. Los objetos estáticos locales se destruyen durante la finalización especificada por `atexit`. Si no se construyó un objeto estático porque el flujo de control del programa omite su declaración, no se intenta destruir ese objeto.

Deducción de tipos en los tipos de valor devueltos (C++14)

En C++14, se puede usar `auto` para indicar al compilador que deduzca el tipo de retorno del cuerpo de la función sin tener que proporcionar un tipo de retorno al final. Tenga en cuenta que `auto` siempre se deduce a un retorno por valor. Use `auto&&` para indicar al compilador que deduzca una referencia.

En este ejemplo, `auto` se deducirá como un valor no-const copia de la suma de `lhs` y `rhs`.

C++

```
template<typename Lhs, typename Rhs>
auto Add2(const Lhs& lhs, const Rhs& rhs)
{
    return lhs + rhs; //returns a non-const object by value
}
```

Tenga en cuenta que `auto` no conserva la const-ness del tipo que deduce. Para las funciones de reenvío cuyo valor de retorno necesita preservar como `const` o referencia de sus argumentos, puede usar la `decltype(auto)` palabra clave, que usa las `decltype` reglas de inferencia de tipos y preserva toda la información de tipos.

`decltype(auto)` Puede usar como valor de retorno normal a la izquierda o como valor devuelto final.

El siguiente ejemplo (basado en el código de [N3493](#)), muestra `decltype(auto)` que se utiliza para permitir el reenvío perfecto de los argumentos de la función en un tipo de retorno que no se conoce hasta que la plantilla es instanciada.

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());
}
```

Devolución de varios valores desde una función

Hay varias formas de devolver más de un valor desde una función:

1. Encapsular los valores en una clase u objeto struct con nombre. Requiere que la definición de la clase o estructura sea visible para el autor de la llamada:

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. Devuelve un objeto std::tuple o std::pair:

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 versión 15.3 y posterior (disponible en modo `/std:c++17` y posterior): Utilizar enlaces estructurados. La ventaja de los enlaces estructurados es que las variables que almacenan los valores devueltos se inicializan al mismo tiempo que se declaran, lo que en algunos casos puede ser significativamente más eficaz. En la declaración `auto[x, y, z] = f();`, los corchetes introducen e inicializan nombres que están en el ámbito de todo el bloque de funciones.

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. Además de utilizar el valor de retorno en sí mismo, puede "devolver" valores definiendo cualquier número de parámetros para utilizar el pasado por referencia de manera que la función pueda modificar o inicializar los valores de los objetos que el autor de la llamada proporciona. Para obtener más información, consulte [Argumentos de la función de tipo referencia](#).

Punteros de función

C++ admite punteros de función de la misma manera que el lenguaje C. Sin embargo, una alternativa con mayor seguridad de tipos suele ser usar un objeto de función.

Se recomienda que use `typedef` para declarar un alias para el tipo de puntero de función si declara una función que devuelve un tipo de puntero de función. Por ejemplo,

C++

```
typedef int (*fp)(int);
fp myFunction(char* s); // function returning function pointer
```

Si esto no se hace, la sintaxis adecuada para la declaración de función se puede deducir de la sintaxis del declarador para el puntero de función reemplazando el identificador (`fp` en el ejemplo anterior) por el nombre de las funciones y la lista de argumentos, como se indica a continuación:

C++

```
int (*myFunction(char* s))(int);
```

La declaración anterior es equivalente a la declaración que usa `typedef` anteriormente.

Consulte también

Sobrecarga de funciones

Funciones con listas de argumentos de variable

Funciones establecidas como valor predeterminado y eliminadas explícitamente

Búsqueda de nombres dependientes de argumentos (Koenig) en las funciones

Argumentos predeterminados

Funciones insertadas

Funciones con listas de argumentos de variable (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

Las declaraciones de función en las que el último miembro son los puntos suspensivos (...) pueden tomar un número variable de argumentos. En estos casos, C++ proporciona comprobación de tipos solo para los argumentos declarados explícitamente. Puede utilizar listas de argumentos variables cuando necesite crear una función tan general que incluso el número y los tipos de argumentos puedan variar. La familia de funciones es un ejemplo de funciones que usan listas de argumentos variables. [printf argument-declaration-list](#)

Funciones con argumentos variables

Para tener acceso a los argumentos declarados en esas funciones, use las macros incluidas en el archivo de inclusión estándar <stdarg.h>, tal como se describe a continuación.

Específicos de Microsoft

Microsoft C++ permite especificar los puntos suspensivos como argumento si son el último argumento y van precedidos por una coma. Por consiguiente, la declaración `int Func(int i, ...);` es válida, pero `int Func(int i ...);` no lo es.

FIN de Específicos de Microsoft

La declaración de una función que toma un número variable de argumentos requiere al menos un argumento de marcador de posición, incluso si no se utiliza. Si no se proporciona este argumento de marcador de posición, no existe ninguna forma de obtener acceso a los argumentos restantes.

Cuando los argumentos de tipo `char` se pasan como argumentos variables, se convierten al tipo `int`. Asimismo, cuando los argumentos de tipo `float` se pasan como argumentos variables, se convierten al tipo `double`. Los argumentos de otros tipos están sujetos a las promociones habituales de entero y de punto flotante. Consulte [Conversiones estándar](#) para obtener más información.

Las funciones que requieren listas de variables se declaran con puntos suspensivos (...) en la lista de argumentos. Use los tipos y macros que se describen en el archivo de inclusión <stdarg.h> para obtener acceso a los argumentos que se pasan por una lista

de variables. Para obtener más información sobre estas macros, consulte [va_arg](#), [va_copy](#), [va_end](#), [va_start](#). en la documentación de la biblioteca en tiempo de ejecución de C.

En el ejemplo siguiente se muestra cómo funcionan las macros con el tipo (declarado en `<stdarg.h>`):

C++

```
// 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 );
            case 'f':
                Printable.f = va_arg( vl, float );
                printf_s( "%f\n", Printable.f );
            case 'c':
                Printable.c = va_arg( vl, char );
                printf_s( "%c\n", Printable.c );
            case 's':
                Printable.s = va_arg( vl, char * );
                printf_s( "%s\n", Printable.s );
        }
    }
}
```

```

        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

```

En el ejemplo anterior se muestran estos conceptos importantes:

1. Debe establecer un marcador de lista como variable de tipo `va_list` antes de que se tenga acceso a cualquier argumento de variable. En el ejemplo anterior, el marcador se denomina `vl`.
2. Se accede a los argumentos individuales mediante la macro `va_arg`. Debe indicar a la macro `va_arg` el tipo de argumento que debe recuperar para poder transferir el número correcto de bytes desde la pila. Si especifica un tipo incorrecto de un tamaño diferente del proporcionado por el programa de llamada a `va_arg`, los resultados son imprevisibles.
3. Debe convertir explícitamente, mediante la macro `va_arg`, el resultado obtenido al tipo que deseé.

Se debe llamar a la macro para finalizar el procesamiento del argumento de variable. `va_end`

Sobrecarga de funciones

Artículo • 03/03/2023 • Tiempo de lectura: 19 minutos

C++ permite especificar más de una función con el mismo nombre en el mismo ámbito. Estas funciones se denominan funciones *sobrecargadas* o *sobrecargas*. Las funciones sobrecargadas permiten proporcionar una semántica diferente para una función, dependiendo de los tipos y el número de argumentos.

Por ejemplo, considere una función `print` que toma un argumento `std::string`. Esta función puede realizar tareas muy diferentes a una función que toma un argumento de tipo `double`. La sobrecarga le impide tener que usar nombres como `print_string` o `print_double`. En tiempo de compilación, el compilador elige qué sobrecarga usar en función de los tipos y el número de argumentos pasados por el autor de la llamada. Si llama a `print(42.0)`, se invoca la función `void print(double d)`. Si llama a `print("hello world")`, se invoca la sobrecarga `void print(std::string)`.

Puede sobrecargar tanto funciones miembro como funciones gratuitas. En la tabla siguiente se muestran las partes de una declaración de función que usa C++ para distinguir entre grupos de funciones con el mismo nombre en el mismo ámbito.

Consideraciones sobre la sobrecarga

| Elemento de declaración de función | ¿Se usa para la sobrecarga? |
|--|--|
| Tipo de valor devuelto de la función | No |
| Número de argumentos | Sí |
| Tipo de argumentos | Sí |
| Presencia o ausencia de puntos suspensivos | Sí |
| Uso de nombres <code>typedef</code> | No |
| Límites de matriz sin especificar | No |
| <code>const</code> o <code>volatile</code> | Sí, cuando se aplica a toda la función |
| Calificadores de referencia (<code>&</code> y <code>const&</code>) | Sí |

Ejemplo

En el ejemplo siguiente se muestra cómo puede usar las sobrecargas de función:

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();
}

```

El código anterior muestra la sobrecarga de la función `print` en el ámbito del archivo.

El argumento predeterminado no se considera parte del tipo de función. Por lo tanto, no se usa en la selección de funciones sobrecargadas. Dos funciones que solo difieren en sus argumentos predeterminados se consideran varias definiciones en lugar de funciones sobrecargadas.

Los argumentos predeterminados no se pueden proporcionar para operadores sobrecargados.

Coincidencia de argumentos

El compilador selecciona la función sobrecargada que se va a invocar en función de la mejor coincidencia entre las declaraciones de función en el ámbito actual a los argumentos proporcionados en la llamada de función. Si se encuentra una función adecuada, se llama a esa función. “Adecuado”, en este contexto, tiene uno de los significados siguientes:

- Se encontró una coincidencia exacta.
- Se realizó una conversión trivial.
- Se realizó una promoción de entero.
- Existe una conversión estándar al tipo de argumento deseado.
- Existe una conversión definida por el usuario (un constructor o un operador de conversión) al tipo de argumento deseado.

- Se encontraron argumentos representados por puntos suspensivos.

El compilador crea un conjunto de funciones de candidato para cada argumento. Las funciones de candidato son funciones en las que el argumento real de esa posición se puede convertir al tipo de argumento formal.

Compila un conjunto de "funciones de coincidencia óptima" para cada argumento y la función seleccionada es la intersección de todos los conjuntos. Si la intersección contiene más de una función, la sobrecarga es ambigua y genera un error. La función seleccionada finalmente siempre es una coincidencia mejor que cada una de las demás funciones del grupo para al menos un argumento. Si no hay ningún ganador claro, la llamada de función genera un error del compilador.

Considere las siguientes declaraciones (las funciones se marcan como `Variant 1`, `Variant 2` y `Variant 3` para su identificación en la siguiente discusión):

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;
```

Considere la instrucción siguiente:

C++

```
F1 = Add( F2, 23 );
```

La instrucción anterior compila dos conjuntos:

Conjunto 1: funciones de candidato cuyo primer argumento es de tipo `Fraction`

Variante 1

Conjunto 2: funciones de candidato cuyo segundo argumento se puede convertir al tipo `int`

Variante 1 (`int` se puede convertir en `long` mediante una conversión estándar)

Variante 3

Las funciones del Conjunto 2 son funciones que tienen conversiones implícitas del tipo de parámetro real al tipo de parámetro formal. Una de esas funciones tiene el "costo" más pequeño para convertir el tipo de parámetro real en su tipo de parámetro formal correspondiente.

La intersección de estos dos conjuntos es Variante 1. Un ejemplo de una llamada de función ambigua es:

C++

```
F1 = Add( 3, 6 );
```

La llamada de función anterior compila los conjuntos siguientes:

| Conjunto 1: Funciones de candidato cuyo primer argumento es de tipo <code>int</code> | Conjunto 2: Funciones de candidato cuyo segundo argumento es de tipo <code>int</code> |
|---|---|
| Variante 2 (<code>int</code> se puede convertir en <code>long</code> mediante una conversión estándar) | Variante 1 (<code>int</code> se puede convertir en <code>long</code> mediante una conversión estándar) |

Dado que la intersección de estos dos conjuntos está vacía, el compilador genera un mensaje de error.

Para la correspondencia de argumento, una función con argumentos predeterminados n se trata como funciones separadas $n+1$, cada una con un número diferente de argumentos.

Los puntos suspensivos (...) actúan como carácter comodín; coinciden con cualquier argumento real. Esto puede conducir a muchos conjuntos ambiguos, si no se diseñan los conjuntos de funciones sobrecargadas con extremo cuidado.

ⓘ Nota

La ambigüedad de las funciones sobrecargadas no se puede determinar hasta que se encuentra una llamada de función. En ese momento, se compilan los conjuntos para cada argumento de la llamada de función y se puede determinar si existe una sobrecarga inequívoca. Esto significa que las ambigüedades pueden permanecer en el código hasta que las evoque una llamada de función determinada.

Diferencias de tipo de argumento

Las funciones sobrecargadas distinguen entre los tipos de los argumentos que toman diferentes inicializadores. Por consiguiente, un argumento de un tipo especificado y una referencia a ese tipo se consideran iguales con el propósito de sobrecarga. Se consideran iguales porque toman los mismos inicializadores. Por ejemplo, `max(double,`

`double`) se considera igual que `max(double &, double &)`. Declarar dos funciones de este tipo produce un error.

Por la misma razón, los argumentos de función de un tipo modificado por `const` o `volatile` no se tratan de manera diferente al tipo base con el propósito de sobrecarga.

Pero, el mecanismo de sobrecarga de función puede distinguir entre las referencias que están calificadas por `const` y `volatile`, y las referencias al tipo base. Así, código como el siguiente es posible:

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& ).
```

Output

Output

```
Over default constructor
Over&
Over default constructor
const Over&
Over default constructor
volatile Over&
```

Los punteros a objetos `const` y `volatile` también se consideran diferentes de los punteros al tipo base con el propósito de sobrecarga.

Coincidencia y conversión de argumentos

Cuando el compilador intenta buscar coincidencias entre argumentos reales y los argumentos de las declaraciones de función, puede proporcionar conversiones estándar o definidas por el usuario para obtener el tipo correcto si no se encuentra ninguna coincidencia exacta. La aplicación de conversiones está sujeta a estas reglas:

- No se tienen en cuenta las secuencias de conversiones que contienen más de una conversión definida por el usuario.
- No se tienen en cuenta las secuencias de conversiones que pueden acortarse quitando las conversiones intermedias.

La secuencia resultante de las conversiones, si existe, se denomina la *mejor coincidencia de secuencia*. Hay varias maneras de convertir un objeto de tipo `int` al tipo `unsigned long` mediante conversiones estándar (descritas en [Conversiones estándar](#)):

- Convierte `int` a `long` y, después, `long` a `unsigned long`.
- Convierte `int` a `unsigned long`.

Aunque la primera secuencia logra el objetivo deseado, no es la mejor coincidencia de secuencia porque existe otra más corta.

En la tabla siguiente se muestra un grupo de conversiones denominado *Conversiones triviales*. Las conversiones triviales tienen un efecto limitado en la secuencia que el compilador elige como la mejor coincidencia. El efecto de las conversiones triviales se describe debajo de la tabla.

Conversiones triviales

| Tipo de argumento | Tipo convertido |
|---------------------------------------|--|
| <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> |

| Tipo de argumento | Tipo convertido |
|-------------------------|----------------------------------|
| <code>type-name*</code> | <code>const type-name*</code> |
| <code>type-name*</code> | <code>volatile type-name*</code> |

Las conversiones se intentan en la siguiente secuencia:

1. Coincidencia exacta. Una coincidencia exacta entre los tipos con los que se llama a la función y los tipos declarados en el prototipo de función siempre es la mejor coincidencia. Las secuencias de conversiones triviales se clasifican como coincidencias exactas. Pero, las secuencias que no realizan ninguna de estas conversiones se consideran mejores que las secuencias que lo hacen:

- De puntero, a puntero a `const` (`type-name*` a `const type-name*`).
- De puntero, a puntero a `volatile` (`type-name*` a `volatile type-name*`).
- De referencia, a referencia a `const` (`type-name&` a `const type-name&`).
- De referencia, a referencia a `volatile` (`type-name&` a `volatile type&`).

2. Coincidencia mediante promociones. Cualquier secuencia no clasificada como coincidencia exacta que solo contiene promociones de enteros, conversiones de `float` a `double`, y conversiones triviales se clasifica como coincidencia mediante promociones. Aunque no es una coincidencia tan buena como cualquier coincidencia exacta, una coincidencia mediante promociones es mejor que una coincidencia mediante conversiones estándar.

3. Coincidencia mediante conversiones estándar. Cualquier secuencia no clasificada como coincidencia exacta o una coincidencia mediante promociones que contenga solo conversiones estándar y conversiones triviales se clasifica como coincidencia mediante conversiones estándar. Dentro de esta categoría, se aplican las reglas siguientes:

- La conversión de un puntero a una clase derivada, a un puntero a una clase base directa o indirecta, es preferible a la conversión a `void *` o a `const void *`.
- La conversión de un puntero a una clase derivada, a un puntero a una clase base, genera una coincidencia mejor cuanto más cerca esté la clase base de una clase base directa. Supongamos que la jerarquía de clases se muestra en la ilustración siguiente:



Gráfico en el que se muestran las conversiones preferidas.

La conversión del tipo D^* al tipo C^* es preferible a la conversión del tipo D^* al tipo B^* .

De forma similar, la conversión del tipo D^* al tipo B^* es preferible a la conversión del tipo D^* al tipo A^* .

Esta regla se aplica también a las conversiones de referencia. La conversión del tipo $D&$ al tipo $C&$ es preferible a la conversión del tipo $D&$ al tipo $B&$, y así sucesivamente.

Esta regla se aplica también a las conversiones de puntero a miembro. La conversión del tipo $T\ D:::^*$ al tipo $T\ C:::^*$ es preferible a la conversión del tipo $T\ D:::^*$ al tipo $T\ B:::^*$, y así sucesivamente, donde T es el tipo del miembro.

La regla anterior solo se aplica a lo largo de una ruta de derivación determinada.

Considere el gráfico que se muestra en la ilustración siguiente.

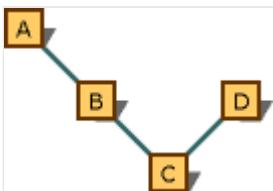


Gráfico de herencia múltiple que muestra las conversiones preferidas.

La conversión del tipo C^* al tipo B^* es preferible a la conversión del tipo C^* al tipo A^* .

La razón es que están en la misma ruta y B^* está más cerca. Pero, la conversión del tipo C^* al tipo D^* no es preferible a la conversión al tipo A^* ; no hay ninguna preferencia, porque las conversiones siguen diferentes rutas.

1. Coincidencia con conversiones definidas por el usuario. Esta secuencia no se puede clasificar como coincidencia exacta, coincidencia mediante promociones o coincidencia mediante conversiones estándar. Para clasificarse como coincidencia con conversiones definidas por el usuario, la secuencia solo debe contener conversiones definidas por el usuario, conversiones estándar o conversiones triviales. Una coincidencia con conversiones definidas por el usuario se considera una coincidencia mejor que una coincidencia con puntos suspensivos (...), pero no tan buena como una coincidencia con conversiones estándar.

2. Coincidencia con puntos suspensivos. La secuencia que coincide con puntos suspensivos en la declaración se clasifica como coincidencia con puntos suspensivos. Se considera la coincidencia más débil.

Se aplican las conversiones definidas por el usuario si no existe ninguna promoción o conversión integrada. Estas conversiones se seleccionan en función del tipo de argumento que se empareja. Observe el código siguiente:

C++

```
// argument_matching1.cpp
class UDC
{
public:
    operator int()
    {
        return 0;
    }
    operator long();
};

void Print( int i )
{
}

UDC udc;

int main()
{
    Print( udc );
}
```

Las conversiones definidas por el usuario disponibles para la clase `UDC` son de tipo `int` y tipo `long`. Por consiguiente, el compilador considera las conversiones para el tipo de objeto que se coteja: `UDC`. Existe una conversión a `int`, que se selecciona.

Durante el proceso de coincidencia de argumentos, las conversiones estándar se pueden aplicar tanto al argumento como al resultado de una conversión definida por el usuario. Por consiguiente, el código siguiente funciona:

C++

```
void LogToFile( long l );
...
UDC udc;
LogFile( udc );
```

En este ejemplo, el compilador invoca una conversión definida por el usuario, `operator long`, para convertir `udc` al tipo `long`. Si no se definió ninguna conversión al tipo `long` definida por el usuario, el compilador primero convertirá el tipo `UDC` al tipo `int` mediante la conversión `operator int` definida por el usuario. Después, se habría aplicado la conversión estándar del tipo `int` al tipo `long` para la coincidencia con el argumento en la declaración.

Si es necesario que las conversiones definidas por el usuario coincidan con un argumento, no se usan las conversiones estándar al evaluar la mejor coincidencia. Incluso si varias funciones candidatas requieren una conversión definida por el usuario, las funciones se consideran iguales. Por ejemplo:

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 );
}
```

Ambas versiones de `Func` requieren una conversión definida por el usuario para convertir el tipo `int` al argumento de tipo de clase. Las conversiones posibles son:

- Conversión del tipo `int` al tipo `UDC1` (conversión definida por el usuario).
- Conversión del tipo `int` al tipo `long`; después, al tipo `UDC2` (conversión en dos pasos).

Aunque el segundo de ellos requiere tanto una conversión estándar, como la conversión definida por el usuario, las dos conversiones todavía se consideran iguales.

ⓘ Nota

Las conversiones definidas por el usuario se consideran conversión por construcción o conversión por inicialización. El compilador considera ambos métodos iguales cuando determina la mejor coincidencia.

Coincidencia de argumentos y el puntero `this`

Las funciones miembro de clase se tratan de manera diferente, dependiendo de si se declaran como `static`. Las funciones `static` no tienen un argumento implícito que proporciona el puntero `this`, por lo que se considera que tienen un argumento menor que las funciones miembro normales. De lo contrario, se declaran de forma idéntica.

Las funciones miembro que no son `static` requieren que el puntero implícito `this` coincida con el tipo de objeto mediante al que se llama a la función. O, para los operadores sobrecargados, requieren que el primer argumento coincida con el objeto al que se aplica el operador. Para más información sobre operadores sobrecargados, vea [Sobrecarga de operadores](#).

A diferencia de otros argumentos en funciones sobrecargadas, el compilador no introduce ningún objeto temporal y no realiza ninguna conversión cuando intenta que coincida el argumento del puntero `this`.

Cuando se usa el operador de selección de miembro `->` para tener acceso a una función miembro de clase `class_name`, el argumento del puntero `this` tiene un tipo de `class_name * const`. Si los miembros se declaran como `const` o `volatile`, los tipos son `const class_name * const` y `volatile class_name * const` respectivamente.

El operador de selección de miembro `.` funciona exactamente de la misma manera, salvo que se antepone como prefijo un operador `&` (address-of) implícito al nombre de objeto. En el ejemplo siguiente se muestra cómo funciona:

C++

```
// Expression encountered in code
obj.name

// How the compiler treats it
(&obj)->name
```

El operando izquierdo de los operadores `->*` y `.*` (puntero a miembro) se trata del mismo modo que los operadores `.` y `->` (selección de miembro) en cuanto a la coincidencia de argumentos.

Calificadores de referencia en funciones miembro

Los calificadores de referencia permiten sobrecargar una función miembro dependiendo de si el objeto al que apunta `this` es rvalue o lvalue. Use esta característica para evitar operaciones de copia innecesarias en escenarios en los que decide no proporcionar acceso de puntero a los datos. Por ejemplo, suponga que la clase `C` inicializa algunos datos en su constructor y devuelve una copia de esos datos en la función miembro `get_data()`. Si un objeto de tipo `C` es un rvalue que está a punto de destruirse, el compilador elige la sobrecarga `get_data() &&`, que mueve los datos en lugar de copiarlos.

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;
}
```

Restricciones en la sobrecarga

Varias restricciones rigen un conjunto aceptable de funciones sobrecargadas:

- Dos funciones cualesquiera de un conjunto de funciones sobrecargadas deben tener distintas listas de argumentos.
- La sobrecarga de funciones con listas de argumentos de los mismos tipos, basándose solo en el tipo de valor devuelto, es un error.

Específicos de Microsoft

Puede sobrecargar `operator new` basándose en el tipo de valor devuelto, en concreto, en el modificador de modelo de memoria especificado.

FIN de Específicos de Microsoft

- Las funciones miembro no se pueden sobrecargar solo porque una es `static` y la otra no es `static`.
- Las declaraciones `typedef` no definen nuevos tipos; presentan sinónimos para tipos existentes. No afectan al mecanismo de sobrecarga. Observe el código siguiente:

C++

```
typedef char * PSTR;

void Print( char *szToPrint );
void Print( PSTR szToPrint );
```

Las dos funciones anteriores tienen listas de argumentos idénticas. `PSTR` es un sinónimo para tipo `char *`. En el ámbito del miembro, este código genera un error.

- Los tipos enumerados son tipos distintos y se pueden utilizar para diferenciar funciones sobrecargadas.
- Los tipos “matriz de” y “puntero a” se consideran idénticos para el propósito de la distinción entre funciones sobrecargadas, pero solo en el caso de matrices

unidimensionales. Las funciones sobrecargadas siguientes están en conflicto y generan un mensaje de error:

C++

```
void Print( char *szToPrint );
void Print( char szToPrint[] );
```

Para matrices con dimensiones más altas, la segunda y todas las dimensiones posteriores se consideran parte del tipo. Se usan en la distinción entre funciones sobrecargadas:

C++

```
void Print( char szToPrint[] );
void Print( char szToPrint[][7] );
void Print( char szToPrint[][9][42] );
```

Sobrecarga, invalidación y ocultación

Dos declaraciones de función cualesquiera con el mismo nombre en el mismo ámbito pueden hacer referencia a la misma función o a dos funciones discretas sobrecargadas. Si las listas de argumentos de las declaraciones contienen argumentos de tipos equivalentes (como se describe en la sección anterior), las declaraciones de función hacen referencia a la misma función. Si no, hacen referencia a dos funciones diferentes que se seleccionan mediante la sobrecarga.

El ámbito de clase se observa estrictamente. Una función declarada en una clase base no está en el mismo ámbito que una función declarada en una clase derivada. Si una función de una clase derivada se declara con el mismo nombre que una función `virtual` en la clase base, la función de la clase derivada *invalida* la función de la clase base. Para más información, vea [Funciones virtuales](#).

Si la función de clase base no se declara como `virtual`, se dice que la función de clase derivada la *oculta*. La invalidación y la ocultación son distintas de la sobrecarga.

El ámbito de bloque se observa estrictamente. Una función declarada en el ámbito del archivo no está en el mismo ámbito que una función declarada de forma local. Si una función declarada localmente tiene el mismo nombre que una función declarada en el ámbito del archivo, la función declarada localmente oculta la función del ámbito del archivo, en lugar de producir una sobrecarga. Por ejemplo:

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" );
}

```

En el código anterior se muestran dos definiciones de la función `func`. La definición que acepta un argumento de tipo `char *` es local para `main` debido a la instrucción `extern`. Por lo tanto, la definición que acepta un argumento de tipo `int` está oculta y la primera llamada a `func` produce un error.

Para funciones miembro sobrecargadas, diferentes versiones de la función pueden recibir diferentes privilegios de acceso. Continúan considerándose en el ámbito de la clase envolvente y, por lo tanto, son funciones sobrecargadas. Considere el código siguiente, en el que se sobrecarga la función miembro `Deposit`; una versión es pública y la otra privada.

El propósito de este ejemplo es proporcionar una clase `Account` que requiera una contraseña correcta para realizar depósitos. Esto se hace mediante la sobrecarga.

La llamada a `Deposit` en `Account::Deposit` llama a la función miembro privada. Esta llamada es correcta porque `Account::Deposit` es una función miembro, y tiene acceso a los miembros privados de la clase.

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;
}

```

Consulte también

[Funciones \(C++\)](#)

Funciones establecidas como valor predeterminado y eliminadas explícitamente

Artículo • 26/09/2022 • Tiempo de lectura: 7 minutos

En C++11, las funciones establecidas como valor predeterminado y eliminadas proporcionan un control explícito sobre si las funciones miembro especiales se generan automáticamente. Las funciones eliminadas también proporcionan un lenguaje simple para impedir que se realicen promociones de tipo problemáticas en argumentos de funciones de todos los tipos (funciones miembro especiales, así como funciones miembro normales y funciones no miembro) que podrían provocar una llamada a función no deseada.

Ventajas de las funciones establecidas como valor predeterminado o eliminadas explícitamente

En C++, el compilador genera automáticamente el constructor predeterminado, el constructor de copias, el operador de asignación de copia y el destructor de un tipo si este no declara los suyos propios. Estas funciones se conocen como *funciones miembro especiales* y son los que hacen que los tipos simples definidos por el usuario en C++ se comporten como las estructuras en C. Es decir, se pueden crear, copiar y destruir sin ningún esfuerzo de codificación adicional. C++11 aporta semántica de movimiento al lenguaje y agrega el constructor de movimiento y el operador de asignación de movimiento a la lista de funciones miembro especiales que el compilador puede generar automáticamente.

Esto es útil en el caso de tipos simples, pero los tipos complejos suelen definir una o varias funciones miembro especiales por sí mismos, lo que puede impedir la generación automática de otras funciones miembro especiales. En la práctica:

- Si se declara explícitamente un constructor, no se genera automáticamente ningún constructor predeterminado.
- Si se declara explícitamente un destructor virtual, no se genera automáticamente ningún destructor predeterminado.

- Si se declara explícitamente un constructor de movimiento o un operador de asignación de movimiento, entonces:
 - No se genera automáticamente ningún constructor de copia.
 - No se genera automáticamente ningún operador de asignación de copia.
- Si se declara explícitamente un constructor de copia, un operador de asignación de copia, un constructor de movimiento, un operador de asignación de movimiento o un destructor, entonces:
 - No se genera automáticamente ningún constructor de movimiento.
 - No se genera automáticamente ningún operador de asignación de movimiento.

ⓘ Nota

Además, el estándar C++11 especifica las reglas adicionales siguientes:

- Si se declara explícitamente un constructor de copia o un destructor, la generación automática del operador de asignación de copia está desusada.
- Si se declara explícitamente un operador de asignación de copia o un destructor, la generación automática del constructor de copia está en desuso.

En ambos casos, Visual Studio sigue generando automáticamente las funciones necesarias de forma implícita y no emite ninguna advertencia.

Las consecuencias de estas reglas también pueden propagarse a las jerarquías de objetos. Por ejemplo, si por cualquier motivo una clase base no puede tener un constructor predeterminado al que se pueda llamar desde una clase derivada (es decir, un constructor `public` o `protected` que no toma ningún parámetro), una clase derivada de ella no puede generar automáticamente su propio constructor predeterminado.

Estas reglas pueden complicar la implementación de lo que deberían ser tipos sencillos definidos por el usuario y expresiones comunes de C++, como la creación de un tipo definido por el usuario que no se puede copiar declarando de forma privada el constructor de copia y el operador de asignación de copia y no definiéndolos.

C++

```
struct noncopyable
{
    noncopyable() {};
```

```
private:  
    noncopyable(const noncopyable&);  
    noncopyable& operator=(const noncopyable&);  
};
```

Antes de C++11, este fragmento de código era la forma idiomática de los tipos que no se pueden copiar. Sin embargo, plantea varios problemas:

- El constructor de copia debe declararse de forma privada para ocultarlo, pero como se ha declarado plenamente, se impide la generación automática del constructor predeterminado. Tiene que definir explícitamente el constructor predeterminado si desea uno, aunque no haga nada.
- Aunque el constructor predeterminado definido de forma explícita no realice ninguna acción, el compilador lo considera no trivial. Es menos eficaz que un constructor predeterminado generado automáticamente e impide que `noncopyable` sea un verdadero tipo POD.
- Aunque el constructor de copia y el operador de asignación de copia estén ocultos para el código externo, las funciones miembro y los elementos friend de `noncopyable` aún pueden verlos y llamarlos. Si se han declarado pero no se han definido, al llamarlos se produce un error del vinculador.
- Aunque se trata de una expresión normalmente aceptada, la intención no está clara a menos que entienda todas las reglas de la generación automática de las funciones miembro especiales.

En C++11, la expresión que no se puede copiar se puede implementar de manera más sencilla.

C++

```
struct noncopyable  
{  
    noncopyable() =default;  
    noncopyable(const noncopyable&) =delete;  
    noncopyable& operator=(const noncopyable&) =delete;  
};
```

Observe cómo se resuelven los problemas con la expresión anterior a C++11:

- La generación del constructor predeterminado todavía se puede evitar declarando el constructor de copia, pero se puede volver a utilizar si se establece explícitamente como valor predeterminado.

- Las funciones miembro especiales establecidas como valor predeterminado explícitamente todavía se consideran triviales, por lo que no hay ninguna reducción del rendimiento y no se impide que `noncopyable` sea un verdadero tipo POD.
- El constructor de copia y el operador de asignación de copia son públicos pero se han eliminado. Es un error en tiempo de compilación definir o llamar a una función eliminada.
- La intención queda clara para cualquiera que entienda `=default` y `=delete`. No es necesario comprender las reglas de generación automática de funciones miembro especiales.

Existen expresiones similares para crear tipos definidos por el usuario que son no móviles, que solo pueden asignarse dinámicamente o que no se pueden asignar dinámicamente. Cada una de estas expresiones tiene implementaciones previas a C++11 que experimentan problemas similares, y que se resuelven de manera similar en C++11 mediante su implementación basada en funciones miembro especiales como valores predeterminados y eliminadas.

Funciones establecidas como valor predeterminado explícitamente

Puede establecer como valor predeterminado cualquiera de las funciones miembro especiales para establecer explícitamente que la función miembro especial usa la implementación predeterminada, definir la función miembro especial con un calificador de acceso no público o restablecer una función miembro especial cuya generación automática no pudo realizarse debido a otras circunstancias.

Una función miembro especial se establece como predeterminada declarándola como en este ejemplo:

```
C++

struct widget
{
    widget()=default;

    inline widget& operator=(const widget&);

    inline widget& widget::operator=(const widget&) =default;
}
```

Tenga en cuenta que puede establecer como valor predeterminado una función miembro especial fuera del cuerpo de una clase siempre y cuando se pueda insertar.

Debido a las ventajas de rendimiento que ofrecen las funciones miembro especiales triviales, se recomienda elegir funciones miembro especiales generadas automáticamente en lugar de cuerpos de función vacíos cuando se desee el comportamiento predeterminado. Se puede hacer si se establece explícitamente como valor predeterminado la función miembro especial o si no la declara (y tampoco declara otras funciones miembro especiales que impedirían que se generara automáticamente).

Funciones eliminadas

Es posible eliminar funciones miembro especiales, así como funciones miembro normales y funciones no miembro, para evitar que se definan o se llamen. La eliminación de funciones miembro especiales proporciona una forma más limpia de impedir que el compilador genere funciones miembro especiales que no se desean. La función se debe eliminar en cuanto se declara; no se puede eliminar después de la manera en que se puede declarar una función y establecerla como valor predeterminado más adelante.

C++

```
struct widget
{
    // deleted operator new prevents widget from being dynamically allocated.
    void* operator new(std::size_t) = delete;
};
```

La eliminación de funciones miembro normales o funciones no miembro impide que las promociones de tipo problemáticas llamen a una función no deseada. Esto funciona porque las funciones eliminadas siguen participando en la resolución de sobrecargas y proporcionan una mejor coincidencia que la función a la que se puede llamar después de que se promuevan los tipos. La llamada a función se resuelve en la función más específica, pero eliminada, y produce un error del compilador.

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; }
```

En el ejemplo anterior, observe que la llamada a `call_with_true_double_only` utilizando un argumento `float` produciría un error del compilador, pero la llamada a `call_with_true_double_only` con un argumento `int` no; en el caso de `int`, el argumento se promoverá de `int` a `double` y llamará correctamente a la versión `double` de la función, aunque esto no sea lo que se esperaba. Para asegurarse de que cualquier llamada a esta función mediante un argumento que no sea `double` produce un error del compilador, se puede declarar una versión de plantilla de la función que se elimina.

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.
```

Búsqueda de nombres dependientes de argumentos (Koenig) en las funciones

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

El compilador puede utilizar la búsqueda de nombres dependiente de argumentos para encontrar la definición de una llamada de función incompleta. La búsqueda de nombres dependiente de argumentos también se denomina búsqueda de Koenig. El tipo de cada argumento de una función se define dentro de una jerarquía de espacios de nombres, clases, estructuras, uniones o plantillas. Cuando se especifica una llamada de función [postfija](#) incompleta, el compilador busca la definición de función en la jerarquía asociada a cada tipo de argumento.

Ejemplo

En el ejemplo, el compilador observa que la función `f()` toma un argumento `x`. El argumento `x` es de tipo `A::x`, que se define en el espacio de nombres `A`. El compilador busca el espacio de nombres `A` y encuentra una definición para la función `f()` que acepta un argumento de tipo `A::X`.

```
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);  
}
```

Argumentos predeterminados

Artículo • 26/09/2022 • Tiempo de lectura: 2 minutos

En muchos casos, las funciones tienen argumentos que se usan con tan poca frecuencia que un valor predeterminado sería suficiente. Para resolver esto, la capacidad de argumento predeterminado permite especificar solo los argumentos de una función que son significativos en una llamada determinada. Para ilustrar este concepto, considere el ejemplo que se presenta en [Sobrecarga de funciones](#).

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.
```

En muchas aplicaciones, se puede proporcionar un valor predeterminado razonable para `prec`, eliminando la necesidad de dos funciones:

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.
```

La implementación de la función `print` cambia ligeramente para reflejar el hecho de que solo existe una función para el tipo `double`:

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();
}

```

Para invocar la nueva función `print`, use código tal como el siguiente:

C++

```

print( d );      // Precision of 2 supplied by default argument.
print( d, 0 ); // Override default argument to achieve other
// results.

```

Tenga en cuenta los siguientes puntos al utilizar argumentos predeterminados:

- Los argumentos predeterminados solo se usan en las llamadas de función donde se omiten los argumentos de finalización; deben ser los últimos argumentos. Por consiguiente, el código siguiente no es válido:

C++

```
int print( double dvalue = 0.0, int prec );
```

- Un argumento predeterminado no se puede volver a definir en declaraciones posteriores aunque la redefinición sea idéntica al original. Por lo tanto, el siguiente código produce un error:

C++

```

// Prototype for print function.
int print( double dvalue, int prec = 2 );

...
// Definition for print function.
int print( double dvalue, int prec = 2 )
{
    ...
}

```

El problema con este código es que la declaración de función de la definición vuelve a definir el argumento predeterminado para `prec`.

- Declaraciones posteriores pueden agregar argumentos predeterminados adicionales.
- Se puede proporcionar argumentos predeterminados para punteros a funciones. Por ejemplo:

C++

```
int (*pShowIntVal)( int i = 0 );
```

Funciones insertadas (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 9 minutos

La palabra clave `inline` indica al compilador que sustituya el código en la definición de función para cada instancia de una llamada a función.

El uso de funciones insertadas puede agilizar la ejecución del programa porque eliminan la sobrecarga asociada a las llamadas a función. El compilador puede optimizar las funciones ampliadas insertadas de maneras que no están disponibles para las funciones normales.

La sustitución del código alineado se produce a discreción del compilador. Por ejemplo, el compilador no alinearán una función si se usa su dirección o si es demasiado grande para alinearla.

Una función definida en el cuerpo de una declaración de clase es una función insertada implícitamente.

Ejemplo

En la siguiente declaración de clase, el constructor `Account` es una función insertada. Las funciones miembro `GetBalance`, `Deposit` y `Withdraw` no se especifican como `inline`, pero se puede implementar como funciones insertadas.

C++

```
// Inline_Member_Functions.cpp
class Account
{
public:
    Account(double initial_balance) { balance = initial_balance; }
    double GetBalance();
    double Deposit( double Amount );
    double Withdraw( double Amount );
private:
    double balance;
};

inline double Account::GetBalance()
{
    return balance;
}

inline double Account::Deposit( double Amount )
{
    return ( balance += Amount );
}
```

```
}
```

```
inline double Account::Withdraw( double Amount )
```

```
{
```

```
    return ( balance -= Amount );
```

```
}
```

```
int main()
```

```
{
```

```
}
```

ⓘ Nota

En la declaración de clase, las funciones se declaran sin la palabra clave `inline`. La palabra clave `inline` se puede especificar en la declaración de clase; el resultado es el mismo.

Una función de miembro insertada determinada se debe declarar de la misma manera en cada unidad de compilación. Esta restricción hace que las funciones insertadas se comporten como si fuesen funciones con instancias creadas. Además, debe haber exactamente una definición de una función insertada.

Una función miembro de clase utiliza de manera predeterminada una vinculación externa, a menos que una definición de esa función contenga el especificador `inline`.

En el ejemplo anterior se muestra que no es necesario declarar estas funciones explícitamente con el especificador `inline`. El uso de `inline` en la definición de función hace que sea una función insertada. Sin embargo, no puede volver a declarar una función como `inline` después de una llamada a esa función.

`inline`, `__inline` y `__forceinline`

Los especificadores `inline` e `__inline` indican al compilador que inserte una copia del cuerpo de la función en cada lugar donde se llama a la función.

La inserción (denominada *expansión alineada* o *alineación*) solo se realiza si el análisis de costos y beneficios del compilador resulta ser rentable. La expansión alineada minimiza la sobrecarga de las llamadas a función con el costo potencial de un tamaño de código mayor.

La palabra clave `__forceinline` invalida el análisis de costos y beneficios, y deja la decisión en manos del programador. Hay que ser prudentes al utilizar `__forceinline`. El uso indiscriminado de `__forceinline` puede dar lugar a código mayor con unas mejoras

de rendimiento mínimas o, en algunos casos, incluso con pérdidas de rendimiento (debido a la mayor paginación que supone un archivo ejecutable mayor, por ejemplo).

El compilador trata las opciones de expansión insertada y las palabras clave como sugerencias. No se garantiza que las funciones se inserten. No se puede forzar que el compilador inserte una función determinada, incluso con la palabra clave `_forceinline`. Al compilar con `/clr`, el compilador no insertará una función si se aplican atributos de seguridad a la función.

A efectos de compatibilidad con versiones anteriores, `_inline` y `_forceinline` son sinónimos de `_inline` y `_forceinline`, respectivamente, a menos que se especifique la opción del compilador [/Za \(Deshabilitar extensiones de lenguaje\)](#).

La palabra clave `inline` indica al compilador que se prefiere la expansión alineada. Sin embargo, el compilador puede crear una instancia independiente de la función y crear vinculaciones de llamada estándar en lugar de insertar el código. Hay dos casos en los que este comportamiento puede ocurrir:

- Funciones recursivas.
- Funciones a las que se hace referencia mediante un puntero en otra parte de la unidad de traducción.

Estos motivos pueden interferir con la alineación, *igual que otros*, a discreción del compilador; no debe depender del especificador `inline` para que se inserte una función.

En lugar de expandir una función insertada definida en un archivo de encabezado, el compilador puede crearla como una función invocable en más de una unidad de traducción. El compilador marca la función generada para el enlazador para evitar infracciones de una regla de definición (ODR).

Al igual que con las funciones normales, no hay ningún orden definido para la evaluación de argumentos en una función insertada. De hecho, podría ser diferente del orden de evaluación de los argumentos cuando se pasan mediante el protocolo normal de llamadas a función.

La opción `/Ob` de optimización del compilador ayuda a determinar si la expansión de funciones insertadas se produce realmente.

`/LTCG` inserta entre módulos si se solicita en el código fuente o no.

Ejemplo 1

C++

```
// inline_keyword1.cpp
// compile with: /c
inline int max( int a , int b ) {
    if( a > b )
        return a;
    return b;
}
```

Las funciones miembro de una clase se pueden declarar alineadas mediante la palabra clave `inline` o colocando la definición de función dentro de la definición de clase.

Ejemplo 2

C++

```
// inline_keyword2.cpp
// compile with: /EHsc /c
#include <iostream>
using namespace std;

class MyClass {
public:
    void print() { cout << i << ' '; } // Implicitly inline
private:
    int i;
};
```

Específico de Microsoft

La palabra clave `__inline` es equivalente a `inline`.

Incluso con `__forceinline`, el compilador no puede insertar código en todas las circunstancias. El compilador no puede insertar una función si:

- La función o su llamador se compila con `/Ob0` (la opción predeterminada para las compilaciones de depuración).
- La función y el llamador utilizan tipos diferentes de control de excepciones (control de excepciones de C++ en uno y control de excepciones estructurado en otro).
- La función tiene una lista de argumentos de variable.
- La función utiliza el ensamblado insertado, a menos que se compile con `/Ox`, `/O1` o `/O2`.

- La función es recursiva y no tiene `#pragma inline_recursion(on)` establecida. Con la directiva `pragma`, las funciones recursivas se alinean hasta una profundidad predeterminada de 16 llamadas. Para reducir la profundidad de inserción, utilice la directiva `pragma inline_depth`.
- La función es virtual y se llama a la misma de forma virtual. Se pueden insertar llamadas directas a funciones virtuales.
- El programa toma la dirección de la función y la llamada se realiza a través del puntero a la función. Se pueden insertar llamadas directas a funciones cuyas direcciones se han tomado.
- La función está marcada también con el modificador `naked_declspec`.

Si el compilador no puede insertar una función declarada con `__forceinline`, genera una advertencia de nivel 1, excepto cuando:

- La función se compila mediante /Od o /Ob0. No se espera ninguna inserción en estos casos.
- La función se define externamente, en una biblioteca incluida u otra unidad de traducción, o es un destino de llamada virtual o un destino de llamada indirecto. El compilador no puede identificar código no insertado que no se encuentra en la unidad de traducción actual.

Se pueden sustituir las funciones recursivas con código insertado hasta una profundidad especificada por la directiva `pragma inline_depth`, hasta un máximo de 16 llamadas. Después de dicha profundidad, las llamadas a función recursivas se tratan como llamadas a una instancia de la función. La profundidad a la que la heurística de alineación examina las funciones recursivas no puede ser superior a 16. La directiva `pragma inline_recursion` controla la expansión alineada de una función que se está expandiendo actualmente. Vea la opción del compilador [Expansión de funciones insertadas](#) (/Ob) para obtener información relacionada.

FIN de Específicos de Microsoft

Para obtener más información sobre cómo utilizar el especificador `inline`, vea:

- [Funciones insertadas de miembro de clase](#)
- [Definir funciones insertadas de C++ con dllexport y dllimport](#)

Cuándo usar funciones insertadas

Las funciones insertadas son útiles para funciones pequeñas, como el acceso a miembros de datos privados. El propósito principal de estas funciones de "descriptor de acceso" de una o dos líneas es devolver información de estado sobre los objetos. Las funciones cortas son sensibles a la sobrecarga de llamadas a función. Las funciones más largas tardan proporcionalmente menos tiempo en la secuencia de llamada y devolución y se benefician menos de la inserción.

Una clase `Point` se puede definir de la siguiente manera:

C++

```
// when_to_use_inline_functions.cpp
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;
}
int main()
{}
```

Si se da por hecho que la manipulación coordinada es una operación relativamente común en un cliente de esta clase, especificar las dos funciones de descriptor de acceso (`x` y `y` en el ejemplo anterior) como `inline` normalmente evita la sobrecarga en:

- Llamadas de función (incluidos el paso de parámetros y la colocación de la dirección del objeto en la pila)
- Conservación del marco de pila del llamador
- Nueva configuración del marco de pila
- Comunicación de valor devuelto

- Restauración del marco de pila antiguo
- Valor devuelto

Funciones insertadas frente a macros

Las funciones insertadas son similares a las macros, ya que el código de función se expande en el punto de la llamada en tiempo de compilación. Sin embargo, el compilador analiza las funciones insertadas y el preprocesador expande las macros. Como consecuencia, hay varias diferencias importantes:

- Las funciones insertadas siguen todos los protocolos de seguridad de tipos exigidos en funciones normales.
- Las funciones insertadas se especifican utilizando la misma sintaxis que cualquier otra función salvo que incluyen la palabra clave `inline` en la declaración de función.
- Las expresiones pasadas como argumentos a funciones insertadas se evalúan una vez. En algunos casos, las expresiones pasadas como argumentos a macros se pueden evaluar más de una vez.

En el ejemplo siguiente se muestra una macro que convierte las minúsculas en mayúsculas:

C++

```
// inline_functions_macro.c
#include <stdio.h>
#include <conio.h>

#define toupper(a) ((a) >= 'a' && ((a) <= 'z') ? ((a)-('a'-'A')):(a))

int main() {
    char ch;
    printf_s("Enter a character: ");
    ch = toupper( getc(stdin) );
    printf_s( "%c", ch );
}
// Sample Input: xyz
// Sample Output: Z
```

El objeto de la expresión `toupper(getc(stdin))` es que un carácter se deba leer desde el dispositivo de consola (`stdin`) y, si es necesario, se convierta a mayúsculas.

Debido a la implementación de la macro, `getc` se ejecuta una vez para determinar si el carácter es mayor o igual que "a", y una vez para determinar si es menor o igual que "z". Si está en ese rango, `getc` se ejecuta de nuevo para convertir el carácter a mayúsculas. Esto significa que el programa espera dos o tres caracteres cuando, idealmente, debe esperar solo uno.

Las funciones insertadas solucionan el problema descrito anteriormente:

C++

```
// inline_functions_inline.cpp
#include <stdio.h>
#include <conio.h>

inline char toupper( char a ) {
    return ((a >= 'a' && a <= 'z') ? a - ('a' - 'A') : a );
}

int main() {
    printf_s("Enter a character: ");
    char ch = toupper( getc(stdin) );
    printf_s( "%c", ch );
}
```

Output

```
Sample Input: a
Sample Output: A
```

Consulte también

[noinline](#)

[auto_inline](#)

Sobrecarga de operadores

Artículo • 26/09/2022 • Tiempo de lectura: 3 minutos

La palabra clave **operator** declara una función especificando el significado del *símbolo de operador* cuando se aplica a las instancias de una clase. Esto proporciona al operador más de un significado, o lo "sobrecarga". El compilador distingue entre los diferentes significados de un operador examinando los tipos de sus operandos.

Sintaxis

tipo operator símbolo de operador(lista de parámetros)

Comentarios

Se puede redefinir la función de la mayoría de los operadores integrados de forma global o clase a clase. Los operadores sobrecargados se implementan como funciones.

El nombre de un operador sobrecargado es **operator***x*, donde *x* es el operador tal como aparece en la tabla siguiente. Por ejemplo, para sobrecargar el operador de suma, se define una función denominada **operator+**. Del mismo modo, para sobrecargar el operador de suma/asignación, **+=**, se define una función denominada **operator+=**.

Operadores redefinibles

| Operator | Nombre | Tipo |
|----------|--------------------------|--------|
| , | Coma | Binary |
| ! | NOT lógico | Unario |
| != | Desigualdad | Binary |
| % | Módulo | Binary |
| %= | Asignación y módulo | Binary |
| & | AND bit a bit | Binary |
| & | Dirección de | Unario |
| && | Y lógico | Binary |
| &= | Asignación AND bit a bit | Binary |

| Operator | Nombre | Tipo |
|----------|--|--------|
| () | Llamada a función | — |
| () | Operador de conversión | Unario |
| * | Multiplicación | Binary |
| * | Desreferencia de puntero | Unario |
| *= | Asignación y multiplicación | Binary |
| + | Suma | Binary |
| + | Unario más | Unario |
| ++ | Incremento ¹ | Unario |
| += | Asignación y suma | Binary |
| - | Resta | Binary |
| - | Negación unaria | Unario |
| -- | Decremento ¹ | Unario |
| --= | Asignación y resta | Binary |
| -> | Selección de miembro | Binary |
| ->* | Selección de puntero a miembro | Binary |
| / | División | Binary |
| /= | Asignación y división | Binary |
| < | Menor que | Binary |
| << | Desplazamiento a la izquierda | Binary |
| <<= | Asignación y desplazamiento a la izquierda | Binary |
| <= | Menor o igual que | Binary |
| = | Asignación | Binary |
| == | Igualdad | Binary |
| > | Mayor que | Binary |
| >= | Mayor o igual que | Binary |
| >> | Desplazamiento a la derecha | Binary |

| Operator | Nombre | Tipo |
|--------------------------|--|--------|
| >>= | Asignación y desplazamiento a la derecha | Binary |
| [] | Subíndice de matriz | — |
| ^ | OR exclusivo | Binary |
| ^= | Asignación y OR exclusivo | Binary |
| | OR inclusivo bit a bit | Binary |
| = | Asignación y OR inclusivo bit a bit | Binary |
| | O lógico | Binary |
| ~ | Complemento a uno | Unario |
| <code>delete</code> | Eliminar | — |
| <code>new</code> | Nuevo | — |
| operadores de conversión | operadores de conversión | Unario |

¹ Existen dos versiones de los operadores de incremento y decremento unarios: preincremento y postincremento.

Vea [Reglas generales de la sobrecarga de operadores](#) para obtener más información. En los temas siguientes se describen las restricciones de las distintas categorías de operadores sobrecargados:

- [Operadores unarios](#)
- [Operadores binarios](#)
- [Cesión](#)
- [Llamada a función](#)
- [Subíndices](#)
- [Acceso a miembros de clase](#)
- [Incremento y decremento](#)
- [Conversiones de tipos definidos por el usuario](#)

Los operadores que se muestran en la tabla siguiente no se pueden sobrecargar. La tabla incluye los símbolos de preprocesador # y ##.

Operadores no redefinibles

| Operator | Nombre |
|----------|---|
| . | Selección de miembro |
| .* | Selección de puntero a miembro |
| :: | Resolución de ámbito |
| ? : | Condicional |
| # | Conversión de preprocesador en una cadena |
| ## | Concatenación de preprocesadores |

Aunque el compilador suele llamar implícitamente a los operadores sobrecargados cuando se encuentran en el código, se pueden invocar explícitamente de la misma manera que cualquier función miembro o no miembro:

C++

```
Point pt;
pt.operator+( 3 ); // Call addition operator to add 3 to pt.
```

Ejemplo

En el ejemplo siguiente se sobrecarga el operador + para sumar dos números complejos y se devuelve el resultado.

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
```

En esta sección

- [Reglas generales para la sobrecarga de operadores](#)
- [Sobrecargar operadores unarios](#)
- [Operadores binarios](#)
- [Cesión](#)
- [Llamada a función](#)
- [Subíndices](#)
- [Acceso a miembros](#)

Consulte también

[Operadores integrados de C++, precedencia y asociatividad](#)

[Palabras clave](#)

Reglas generales para la sobrecarga de operadores

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

Las reglas siguientes restringen la forma en que se implementan los operadores sobrecargados. Sin embargo, no se aplican a los operadores `new` y `delete`, que se tratan por separado.

- No se pueden definir operadores nuevos, por ejemplo .
- No se puede volver a definir el significado de los operadores cuando se aplican a los tipos de datos integrados.
- Los operadores sobrecargados deben ser una función miembro de clase no estática o una función global. Una función global que necesita acceso a miembros de clase privados o protegidos se debe declarar como friend de esa clase. Una función global debe tomar al menos un argumento que sea de clase o de tipo enumerado, o que sea una referencia a una clase o a un tipo enumerado. Por ejemplo:

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()
{}
```

En el ejemplo de código anterior se declara el operador "menos que" como función miembro; sin embargo, los operadores de suma se declaran como funciones globales con acceso de confianza. Observe que se puede proporcionar más de una implementación para un operador determinado. En el caso del operador de suma anterior, las dos implementaciones se proporcionan para facilitar la propiedad conmutativa. También es probable que puedan implementarse operadores que agregan `Point` a `Point`, `int` a `Point`, etc.

- Los operadores obedecen a la prioridad, la agrupación y el número de operandos dictados por su uso típico con los tipos integrados. Por consiguiente, no hay ninguna forma de expresar el concepto "sumar 2 y 3 a un objeto de tipo `Point`" con la expectativa de que 2 se sume a la coordenada *X* y 3 se sume a la coordenada *Y*.
- Los operadores unarios declarados como funciones miembro no toman ningún argumento; si se declaran como funciones globales, toman un argumento.
- Los operadores binarios declarados como funciones miembro toman un argumento; si se declaran como funciones globales, toman dos argumentos.
- Si un operador puede utilizarse como operador unario u operador binario (`&`, `*`, `+` y `-`), se puede sobrecargar cada uso por separado.
- Los operadores sobrecargados no pueden tener argumentos predeterminados.
- Las clases derivadas heredan todos los operadores sobrecargados, excepto el de asignación (`operator=`).
- El primer argumento para los operadores sobrecargados de función miembro siempre es del tipo de clase del objeto para el que se invoca el operador (la clase en la que se declara el operador o una clase derivada de ella). No se proporciona ninguna conversión para el primer argumento.

Observe que el significado de cualquiera de los operadores se puede cambiar por completo. Esto incluye el significado de los operadores `address-of` (`&`), `assignment` (`=`) y `function-call`. Además, las identidades en las que se puede confiar para los tipos integrados pueden cambiarse mediante la sobrecarga de operadores. Por ejemplo, las cuatro instrucciones siguientes suelen ser equivalentes cuando se evalúan completamente:

C++

```
var = var + 1;
var += 1;
var++;
++var;
```

No se puede confiar en esta identidad para los tipos de clase que sobrecargan operadores. Además, algunos requisitos implícitos en el uso de estos operadores para los tipos básicos se relajan para los operadores sobrecargados. Por ejemplo, el operador de suma o asignación, `+=`, requiere que el operando izquierdo sea un valor L cuando se aplica a los tipos básicos; este requisito no existe cuando se sobrecarga el operador.

Nota

Por coherencia, a menudo es mejor seguir el modelo de los tipos integrados al definir operadores sobrecargados. Si la semántica de un operador sobrecargado difiere mucho de su significado en otros contextos, puede ser más confusa que útil.

Consulte también

[Sobrecarga de operadores](#)

Sobrecargar operadores unarios

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Los operadores unarios generan un resultado desde un único operando. Puede definir sobrecargas de un conjunto estándar de operadores unarios para trabajar en tipos definidos por el usuario.

Operadores unarios sobrecargables

Puede sobrecargar los siguientes operadores unarios en tipos definidos por el usuario:

- `!` (NOT lógico)
- `&` (dirección de)
- `~` (complemento)
- `*` (desreferencia de puntero)
- `+` (unario más)
- `-` (negación unaria)
- `++` (incremento de prefijo) o (incremento de postfijo)
- `--` (decremento de prefijo) o (decremento de postfijo)
- Operadores de conversión

Declaraciones de sobrecarga del operador unario

Puede declarar operadores unarios sobrecargados como funciones miembro no estáticas o como funciones no miembro. Las funciones miembro unarias sobrecargadas no toman ningún argumento porque operan implícitamente en `this`. Las funciones no miembro se declaran con un argumento. Cuando se declaran ambos formularios, el compilador sigue las reglas de la resolución de sobrecargas para determinar qué función se va a usar, si se usa alguna.

Las reglas siguientes se aplican a todos los operadores unarios de prefijo. Para declarar una función de operador unario como una función miembro no estática, use este

formulario de declaración:

```
return-type operator op ();
```

En este formulario, `return-type` es el tipo de valor devuelto y `op` es uno de los operadores enumerados en la tabla anterior.

Para declarar una función de operador unario como una función no miembro, use este formulario de declaración:

```
return-type operator op ( class-type );
```

En este formulario, `return-type` es el tipo de valor devuelto, `op` es uno de los operadores enumerados en la tabla anterior y `class-type` es el tipo de clase del argumento en el que se va a operar.

Las formas de postfijo de `++` y `--` toman un argumento adicional `int` para distinguirlos de los formularios de prefijo. Para obtener más información sobre los formularios de prefijo y postfijo de `++` y `--`, consulte [Sobrecarga de operadores de incremento y decremento](#).

ⓘ Nota

No hay restricciones en los tipos de valor devuelto de los operadores unarios. Por ejemplo, tiene sentido que el operador NOT lógico (`!`) devuelva un valor `bool`, pero este comportamiento no se exige.

Consulte también

[Sobrecarga de operadores](#)

Sobrecarga de operadores de incremento y decremento (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Los operadores de incremento y decremento pertenecen a una categoría especial porque hay dos variantes de cada uno de ellos:

- Preincremento y postincremento
- Predecremento y postdecremento

Cuando escriba funciones de operador sobrecargadas, puede resultarle útil implementar versiones distintas para las versiones de prefijo y de postfixo de estos operadores. Para distinguirlas, se aplica la regla siguiente: la forma de prefijo del operador se declara exactamente del mismo modo que cualquier otro operador unario; la forma de postfixo acepta un argumento adicional de tipo `int`.

ⓘ Nota

Cuando se especifica un operador sobrecargado en la forma de postfixo del operador de incremento o decremento, el argumento adicional debe ser de tipo `int`; la especificación de cualquier otro tipo genera un error.

En el ejemplo siguiente se muestra cómo definir operadores de incremento y decremento de prefijo y de postfixo para la clase `Point`:

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()
{
}

```

Pueden definirse los mismos operadores en el ámbito de archivo (global) mediante los siguientes prototipos de función:

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

```

El argumento de tipo `int` que indica la forma de postfijo del operador de incremento o decremento no suele usarse para pasar argumentos. Normalmente contiene el valor 0. Sin embargo, se puede utilizar del modo siguiente:

```
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.  
}
```

No hay sintaxis alguna para usar los operadores de incremento y decremento para pasar estos valores que no sea la invocación explícita, como se muestra en el código anterior. Una manera más sencilla de implementar esta funcionalidad es sobrecargar el operador de suma/asignación (`+=`).

Consulte también

[Sobrecarga de operadores](#)

Operadores binarios

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

En la tabla siguiente se muestra una lista de operadores que se pueden sobrecargar.

Operadores binarios redefinibles

| Operator | Nombre |
|----------|--|
| , | Coma |
| != | Desigualdad |
| % | Módulo |
| %= | Módulo/asignación |
| & | AND bit a bit |
| && | Y lógico |
| &= | AND bit a bit/asignación |
| * | Multiplicación |
| *= | Multiplicación/asignación |
| + | Suma |
| += | Suma/asignación |
| - | Resta |
| -= | Resta/asignación |
| -> | Selección de miembro |
| ->* | Selección de puntero a miembro |
| / | División |
| /= | División/asignación |
| < | Menor que |
| << | Desplazamiento a la izquierda |
| <<= | Desplazamiento a la izquierda/asignación |

| Operator | Nombre |
|------------------------|--|
| <code><=</code> | Menor o igual que |
| <code>=</code> | Asignación |
| <code>==</code> | Igualdad |
| <code>></code> | Mayor que |
| <code>>=</code> | Mayor o igual que |
| <code>>></code> | Desplazamiento a la derecha |
| <code>>>=</code> | Desplazamiento a la derecha/asignación |
| <code>^</code> | OR exclusivo |
| <code>^=</code> | OR exclusivo/asignación |
| <code> </code> | OR inclusivo bit a bit |
| <code> =</code> | OR inclusivo bit a bit/asignación |
| <code> </code> | O lógico |

Para declarar una función de operador binario como miembro no estático, debe declararla de la forma siguiente:

```
ret-type operator op(arg)
```

donde *ret-type* es el tipo devuelto, *op* es uno de los operadores que aparecen en la tabla anterior y *arg* es un argumento de cualquier tipo.

Para declarar una función de operador binario como función global, debe declararla de la forma siguiente:

```
ret-type operator op(arg1,arg2)
```

donde where *ret-type* y *op* son como se describen para las funciones de operador de miembro y *arg1* y *arg2* son argumentos. Al menos uno de los argumentos debe ser de tipo de clase.

① Nota

No hay restricciones para los tipos de valor devuelto de los operadores binarios; sin embargo, la mayoría de los operadores binarios definidos por el usuario devuelven

un tipo de clase o una referencia a un tipo de clase.

Consulte también

[Sobrecarga de operadores](#)

Asignación

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

El operador de asignación (=) es, en rigor, un operador binario. La declaración es idéntica a cualquier otro operador binario, con las excepciones siguientes:

- Debe ser una función miembro no estática. Ninguna función **operator=** se puede declarar como función no miembro.
- Las clases derivadas no lo heredan.
- El compilador puede generar una función **operator=** predeterminada para los tipos de clase si no existe ninguna.

El ejemplo siguiente muestra cómo declarar un operador de asignación:

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;
}
```

El argumento proporcionado es el lado derecho de la expresión. El operador devuelve el objeto para preservar el comportamiento del operador de asignación, que devuelve el valor del lado izquierdo después de que se complete la asignación. Esto permite encadenar asignaciones, como:

C++

```
pt1 = pt2 = pt3;
```

El operador de asignación de copia no debe confundirse con el constructor de copia. Este último se llama durante la construcción de un nuevo objeto a partir de uno existente:

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.
```

ⓘ Nota

Es aconsejable seguir la [regla de tres](#) de que una clase que define un operador de asignación de copia también debe definir explícitamente el constructor de copia, el destructor y, a partir de C++11, el constructor de movimiento y el operador de asignación de movimiento.

Consulte también

- [Sobrecarga de operadores](#)
- [Constructores de copia y operadores de asignación de copia \(C++\)](#)

Llamada de función (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

El operador de llamada a función, invocado mediante paréntesis, es un operador binario.

Sintaxis

```
primary-expression ( expression-list )
```

Comentarios

En este contexto, `primary-expression` es el primer operando, y `expression-list` (una lista de argumentos que posiblemente esté vacía) es el segundo operando. El operador de llamada a función se utiliza para las operaciones que requieren varios parámetros. Esto funciona porque `expression-list` es una lista en lugar de un solo operando. El operador de llamada a función debe ser una función miembro no estática.

El operador de llamada a función, cuando está sobrecargado, no modifica la forma de llamar a las funciones; en su lugar, modifica cómo debe interpretarse el operador cuando se aplica a objetos de un tipo de clase especificado. Por ejemplo, el código siguiente normalmente no tendría sentido:

C++

```
Point pt;
pt( 3, 2 );
```

Pero, con un operador sobrecargado de llamada a función adecuado, esta sintaxis se puede usar para desplazar 3 unidades la coordenada `x` y 2 unidades la coordenada `y`. En el código siguiente se muestra esa definición:

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 );
}

```

Tenga en cuenta que el operador de llamada a función se aplica al nombre de un objeto, no al nombre de una función.

También puede sobrecargar el operador de llamada de función mediante un puntero a una función (en lugar de a la función en sí).

C++

```

typedef void(*ptf)();
void func()
{
}
struct S
{
    operator ptf()
    {
        return func;
    }
};

int main()
{
    S s;
    s(); //operates as s.operator ptf()()
}

```

Consulte también

[Sobrecarga de operadores](#)

Subíndices

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

El operador de subíndice ([]), igual que el operador de llamada de función, se considera un operador binario. El operador de subíndice debe ser una función miembro no estática que tome un único argumento. Este argumento puede ser de cualquier tipo y designa el subíndice de la matriz deseado.

Ejemplo

En el ejemplo siguiente, se muestra cómo crear un vector de tipo `int` que implementa la comprobación de límites:

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
```

Comentarios

Cuando `i` llega a 10 en el programa anterior, `operator[]` detecta que se está utilizando un subíndice fuera de los límites y emite un mensaje de error.

Observe que la función `operator[]` devuelve un tipo de referencia. Esto lo convierte en un valor L, que permite utilizar expresiones con subíndice en cada lado de los operadores de asignación.

Consulte también

[Sobrecarga de operadores](#)

Acceso a miembros

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

El acceso a miembros de clase se puede controlar mediante la sobrecarga del operador de acceso a miembros (->). Este operador se considera un operador unario en este uso, y la función de operador sobrecargado debe ser una función miembro de clase. Por lo tanto, la declaración de una función de ese tipo es:

Sintaxis

```
class-type *operator->()
```

Comentarios

donde *class-type* es el nombre de la clase a la que pertenece este operador. La función de operador de acceso a miembros debe ser una función miembro no estática.

Este operador se usa (a menudo junto con el operador de desreferenciación de puntero) para implementar "punteros inteligentes" que validan punteros antes del uso de desreferenciación o de recuento.

El operador de acceso a miembros . no se puede sobrecargar.

Consulte también

[Sobrecarga de operadores](#)

Clases y structs (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

En esta sección se presentan las clases y structs de C++. Las dos construcciones son idénticas en C++, salvo que, en los structs, la accesibilidad predeterminada es pública, mientras que en las clases es privada.

Las clases y los structs son las construcciones con las que define sus propios tipos. Las clases y los structs pueden contener miembros de datos y funciones miembro, lo que permite describir el comportamiento y el estado del tipo.

Se tratan los siguientes temas:

- [class](#)
- [struct](#)
- [Información general sobre miembros de clase](#)
- [Control de acceso a miembros](#)
- [Herencia](#)
- [Miembros estáticos](#)
- [Conversiones de tipos definidos por el usuario](#)
- [Miembros de datos mutables \(especificador mutable\)](#)
- [Declaraciones de clase anidadas](#)
- [Tipos de clase anónima](#)
- [Punteros a miembros](#)
- [this \(Puntero\)](#)
- [Campos de bits de C++](#)

Los tres tipos de clase son estructura, clase, y unión. Se declaran mediante las palabras clave [struct](#), [class](#) y [union](#). En la tabla siguiente se muestran las diferencias entre los tres tipos de clase.

Para obtener más información sobre las uniones, consulte [Uniones](#). Para obtener información sobre las clases y estructuras en C++/CLI y C++/CX, consulte [Clases y estructuras](#).

Control de acceso y restricciones de las estructuras, clases y uniones

| Estructuras | Clases | Uniones |
|--|---|---|
| La clave de clase es <code>struct</code> | La clave de clase es <code>class</code> | La clave de clase es <code>union</code> |
| El acceso predeterminado es público | El acceso predeterminado es privado | El acceso predeterminado es público |
| No hay ninguna restricción de uso | No hay ninguna restricción de uso | Usan solo un miembro cada vez |

Vea también

[Referencia del lenguaje C++](#)

class (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

La palabra clave **class** declara un tipo de clase o define un objeto de un tipo de clase.

Sintaxis

```
[template-spec]
class [ms-decl-spec] [tag [: base-list ]]
{
    member-list
} [declarators];
[ class ] tag declarators;
```

Parámetros

template-spec

Especificaciones de plantilla opcionales. Para obtener más información, consulta [Plantillas](#).

class

La palabra clave **class**.

ms-decl-spec

Especificación opcional de clase de almacenamiento. Para más información, consulte la palabra clave [_declspec](#).

tag

Nombre del tipo asignado a la clase. La etiqueta se convierte en una palabra reservada dentro del ámbito de la clase. La etiqueta es opcional. Si se omite, se define una clase anónima. Para obtener más información, consulta [Tipos de clase anónimos](#).

base-list

Lista opcional de clases o de estructuras de las que esta clase derivará sus miembros. Consulta [Clases base](#) para obtener más información. Cada nombre de clase base o de estructura puede ir precedido de un especificador de acceso ([public](#), [private](#), [protected](#)) y la palabra clave [virtual](#). Consulte la tabla de acceso a miembros en [Controlar el acceso a los miembros de clase](#) para más información.

member-list

Lista de miembros de clase. Consulta [Información general sobre miembros de clase](#) para obtener más información.

declarators

Lista de declaradores que especifica los nombres de una o más instancias del tipo de clase. Los declaradores pueden incluir listas de inicializadores si todos los miembros de datos de la clase son `public`. Esto es más frecuente en las estructuras, cuyos miembros de datos son `public` de forma predeterminada, que en las clases. Consulta [Información general sobre declaradores](#) para obtener más información.

Comentarios

Para obtener más información sobre las clases en general, vea uno de los temas siguientes:

- [struct](#)
- [union](#)
- [_multiple_inheritance](#)
- [_single_inheritance](#)
- [_virtual_inheritance](#)

Para obtener información sobre las clases y estructuras en C++/CLI y C++/CX, consulta [Clases y estructuras](#)

Ejemplo

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;
}

```

Consulte también

[Palabras clave](#)

[Clases y structs](#)

struct (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

La palabra clave **struct** define un tipo de estructura o una variable de un tipo de estructura.

Sintaxis

```
[template-spec] struct [ms-decl-spec] [tag [: base-list ]]
{
    member-list
} [declarators];
[struct] tag declarators;
```

Parámetros

template-spec

Especificaciones de plantilla opcionales. Para más información, consulte [Especificaciones de plantilla](#).

struct

La palabra clave **struct**.

ms-decl-spec

Especificación opcional de clase de almacenamiento. Para más información, consulte la palabra clave [_declspec](#).

tag

Nombre del tipo dado a la estructura. La etiqueta se convierte en una palabra reservada dentro del ámbito de la estructura. La etiqueta es opcional. Si se omite, se define una estructura anónima. Para más información, consulte [Tipos de clase anónimos](#).

base-list

La lista opcional de clases o de estructuras de la que esta estructura derivará sus miembros. Consulte [Clases base](#) para más información. Cada nombre de clase base o de estructura puede ir precedido de un especificador de acceso ([public](#), [private](#), [protected](#)) y la palabra clave [virtual](#). Consulte la tabla de acceso a miembros en [Controlar el acceso a los miembros de clase](#) para más información.

member-list

Lista de miembros de la estructura. Consulte [Información general sobre miembros de clase](#) para más información. La única diferencia aquí es que se utiliza `struct` en lugar de `class`.

declarators

Lista de declaradores que especifican los nombres de la estructura. Las listas de declaradores declaran una o más instancias del tipo de estructura. Los declaradores pueden incluir listas de inicializadores si todos los miembros de datos de la estructura son `public`. Las listas de inicializadores son comunes en estructuras porque los miembros de datos son `public` de forma predeterminada. Consulte [Información general sobre declaradores](#) para más información.

Comentarios

Un tipo de estructura es un tipo compuesto definido por el usuario. Se compone de campos o de miembros que pueden tener diferentes tipos.

En C++, una estructura es igual que una clase salvo que sus miembros son `public` de forma predeterminada.

Para obtener información sobre las clases y estructuras administradas en C++/CLI, consulte [Clases y estructuras](#).

Uso de una estructura

En C, debe utilizar explícitamente la palabra clave `struct` para declarar una estructura.

En C++, no es necesario usar la palabra clave `struct` una vez definido el tipo.

Tiene la opción de declarar variables al definir el tipo de estructura, para lo cual debe insertar uno o más nombres de variable separados por comas entre la llave de cierre y el punto y coma.

Las variables de estructura se pueden inicializar. La inicialización de cada variable se debe incluir entre llaves.

Para obtener información relacionada, consulte [clase](#), [unión](#) y [enumeración](#).

Ejemplo

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; // 00000??? 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
```

Información general sobre miembros de clase

Artículo • 03/03/2023 • Tiempo de lectura: 4 minutos

Un `class` o `struct` consta de sus miembros. Las funciones miembro son las encargadas de realizar el trabajo de la clase a la que pertenecen. El estado que mantienen se almacena en sus miembros de datos. La inicialización de los miembros se lleva a cabo mediante constructores y el trabajo de limpieza, como la liberación de memoria y de los recursos se realiza mediante destructores. En C++11 y versiones posteriores, los miembros de datos pueden (y normalmente deberían) inicializarse en el punto en el que se declaran.

Tipos de miembros de clase

La lista completa de categorías de miembros es la siguiente:

- [Funciones miembro especiales.](#)
- [Información general de Funciones miembro.](#)
- Miembros de datos [mutables](#) y [estáticos](#), como los tipos integrados y otros tipos definidos por el usuario.
- Operadores
- [Declaraciones de clase anidadas](#) y.)
- [Uniones](#)
- [Enumeraciones.](#)
- [Campos de bit.](#)
- [Friends.](#)
- [Alias y definiciones de tipos.](#)

ⓘ Nota

Se incluyen Friends en la lista anterior porque están contenidos en la declaración de clase. Sin embargo, no son miembros de clase verdaderos, porque no están en el ámbito de la clase.

Ejemplo de declaración de clase

En el siguiente ejemplo se muestra una declaración de clase sencilla:

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 };
```

Accesibilidad de miembros

Los miembros de una clase se declaran en la lista de miembros. La lista de miembros de una clase se puede dividir en un número cualquiera de secciones `private`, `protected` y

`public` mediante el uso de palabras clave conocidas como especificadores de acceso.

Un signo de dos puntos `:` debe ir a continuación del especificador de acceso. Estas secciones no tienen que ser contiguas, es decir, cualquiera de estas palabras clave puede aparecer varias veces en la lista de miembros. La palabra clave designa el acceso de todos los miembros hacia arriba hasta el especificador de acceso siguiente o la llave de cierre. Para más información, consulte [Control de acceso a miembros \(C++\)](#).

Miembros estáticos

Un miembro de datos se puede declarar como `static`, lo que significa que todos los objetos de la clase tienen acceso a la misma copia del mismo. Una función miembro puede declararse como `static`, en cuyo caso solo puede tener acceso a los miembros de datos estáticos de la clase (y no tiene puntero `this`). Para más información, consulte [Miembros de datos estáticos](#).

Funciones miembro especiales

Las funciones miembro especiales son funciones que el compilador proporciona automáticamente si no las especifica en el código fuente.

- Constructor predeterminado
- Constructor de copias
- (C++11) Constructor de movimiento
- Operador de asignación de copia
- (C++11) Operador de asignación de movimiento
- Destructor

Para más información, consulte [Funciones miembro especiales](#).

Inicialización miembro a miembro

En C++11 y versiones posteriores, los declaradores de miembros no estáticos pueden contener inicializadores.

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()
{
}

```

Si un miembro recibe un valor en un constructor, ese valor sobrescribe el valor asignado en la declaración.

Solo hay una copia compartida de los miembros de datos estáticos para todos los objetos de un tipo de clase determinado. Los miembros de datos estáticos se deben definir y se pueden inicializar en el ámbito de archivo. Para más información sobre los miembros de datos estáticos, consulte [Miembros de datos estáticos](#). En el ejemplo siguiente se muestra cómo inicializar miembros de datos estáticos:

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;

```

 Nota

El nombre de clase, `CanInit2`, debe preceder a `i` para especificar que el `i` definido es un miembro de la clase `CanInit2`.

Consulte también

[Clases y structs](#)

Control de acceso a miembros (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 7 minutos

Los controles de acceso permiten separar la `public` interfaz de una clase de los `private` detalles de implementación y de los `protected` miembros que solo pueden usar las clases derivadas. El especificador de acceso se aplica a todos los miembros declarados después de él hasta que se encuentra el especificador de acceso siguiente.

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();
};
```

El acceso predeterminado es `private` en una clase, y `public` en una estructura o unión. Los especificadores de acceso de una clase se pueden usar cualquier número de veces en cualquier orden. La asignación de almacenamiento para los objetos de los tipos de clase depende de la implementación. Sin embargo, los compiladores deben garantizar la asignación de miembros a direcciones de memoria sucesivamente superiores entre los especificadores de acceso.

Control de acceso a miembros

| Tipo de Acceso | Significado |
|------------------------|---|
| <code>private</code> | Los miembros de la clase declarados como <code>private</code> solo pueden ser usados por las funciones miembro y amigos (clases o funciones) de la clase. |
| <code>protected</code> | Los miembros de la clase declarados como <code>protected</code> pueden ser usados por las funciones miembro y amigos (clases o funciones) de la clase. Además, las clases derivadas de la clase también pueden usarlos. |

| Tipo de Acceso | Significado |
|---------------------|---|
| <code>public</code> | Los miembros de la clase declarados como <code>public</code> pueden ser usados por cualquier función. |

El control de acceso ayuda a evitar que se utilicen los objetos de forma no prevista. Esta protección se pierde cuando se realizan conversiones de tipo explícitas (conversiones).

ⓘ Nota

El control de acceso también es aplicable a todos los nombres: funciones miembro, datos de miembro, clases anidadas y enumeradores.

Control de acceso en clases derivadas

Dos factores controlan los miembros de una clase base que están accesibles en una clase derivada; estos mismos factores controlan el acceso a los miembros heredados en la clase derivada:

- Si la clase derivada declara la clase base usando el `public` especificador de acceso.
- Que el acceso al miembro esté en la clase base.

En la tabla siguiente se muestra la interacción entre estos factores y cómo se determina el acceso a miembros de clase base.

Acceso a miembros de clase base

| <code>private</code> | <code>protected</code> | <code>public</code> |
|--|---|---|
| Siempre inaccesible con cualquier acceso de derivación | <code>private</code> en la clase derivada si se utiliza <code>private</code> derivación | <code>private</code> en la clase derivada si se utiliza <code>private</code> derivación |
| | <code>protected</code> en la clase derivada si se utiliza <code>protected</code> derivación | <code>protected</code> en la clase derivada si se utiliza <code>protected</code> derivación |
| | <code>protected</code> en la clase derivada si se utiliza <code>public</code> derivación | <code>public</code> en la clase derivada si se utiliza <code>public</code> derivación |

El siguiente ejemplo ilustra la derivación del acceso:

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
}
```

En `DerivedClass1`, la función miembro `PublicFunc` es un `public` miembro y `ProtectedFunc` es una `protected` miembro porque `BaseClass` es una `public` clase base. `PrivateFunc` es `private` a `BaseClass`, y es inaccessible para cualquier clase derivada.

En `DerivedClass2`, las funciones `PublicFunc` y `ProtectedFunc` se consideran `private` miembros porque `BaseClass` es una `private` clase base. De nuevo, `PrivateFunc` es `private` a `BaseClass`, y es inaccessible para cualquier clase derivada.

Puede declarar una clase derivada sin un especificador de acceso de clase base. En este caso, la derivación se considera **private** si la declaración de la clase derivada utiliza la **class** palabra clave. La derivación se considera **public** si la declaración de la clase derivada utiliza la **struct** palabra clave. Por ejemplo, el código siguiente:

C++

```
class Derived : Base  
...
```

equivale a:

C++

```
class Derived : private Base  
...
```

De igual forma, el código siguiente:

C++

```
struct Derived : Base  
...
```

equivale a:

C++

```
struct Derived : public Base  
...
```

Los miembros declarados con **private** acceso no son accesibles a las funciones o clases derivadas a menos que dichas funciones o clases se declaren usando la **friend** declaración de la clase base.

Un tipo **union** no puede tener una clase base.

ⓘ Nota

Cuando se especifica una private clase base, es aconsejable utilizar explícitamente la **private** palabra clave para que los usuarios de la clase derivada reconozcan el acceso de miembros.

Control de acceso y miembros estáticos

Cuando se especifica una clase base como `private`, solo afecta a los miembros no estáticos. Los miembros estáticos públicos siguen siendo accesibles en las clases derivadas. Sin embargo, el acceso a los miembros de la clase base mediante punteros, referencias u objetos puede requerir una conversión, que aplica de nuevo el control de acceso. Considere el ejemplo siguiente:

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;
}
```

En el código anterior, el control de acceso prohíbe la conversión de un puntero a `Derived2` en un puntero a `Base`. El puntero `this` es implícitamente de tipo `Derived2 *`. Para seleccionar la `CountOf` función, `this` debe convertirse en tipo `Base *`. Esta conversión no está permitida porque `Base` es una private clase base indirecta a `Derived2`. La conversión a un private tipo de clase base solo es aceptable para los

punteros a clases derivadas inmediatas. Por ello los punteros de tipo `Derived1 *` pueden ser convertidos a tipo `Base *`.

Una llamada explícita a la función `CountOf`, sin utilizar un puntero, referencia u objeto para seleccionarla, no implica ninguna conversión. Por eso se permite la llamada.

Los miembros y amigos de una clase derivada, `T`, pueden convertir un puntero a `T` en un puntero a una private clase base directa de `T`.

Acceso a funciones virtuales

El control de acceso aplicado a las funciones `virtual` viene determinado por el tipo utilizado para realizar la llamada a la función. Las declaraciones de anulación de la función no afectan al control de acceso para un tipo determinado. Por ejemplo:

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;
}
```

En el ejemplo anterior, la llamada a la función virtual `GetState` utilizando un puntero al tipo `VFuncBase` llama `VFuncDerived::GetState`, y `GetState` se trata como `public`. Sin

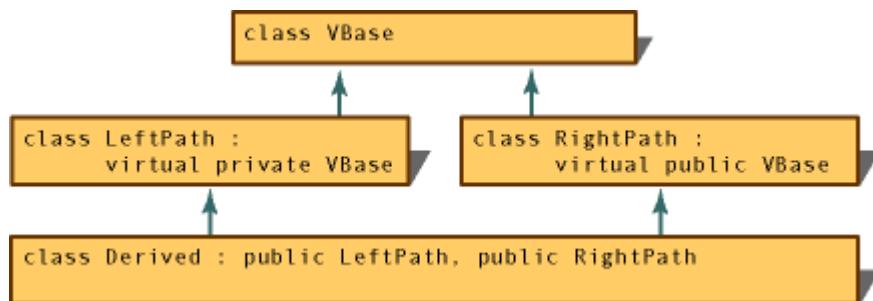
embargo, llamar `GetState` usando un puntero al tipo `VFuncDerived` es una violación de control de acceso porque `GetState` se declara `private` en la clase `VFuncDerived`.

⊗ Precaución

La función virtual `GetState` se puede llamar mediante un puntero a la clase base `VFuncBase`. Esto no significa que la función llamada sea la versión de la clase base de esa función.

Control de acceso con herencia múltiple

En los entramados de herencia múltiple con clases base virtuales, se puede acceder a un nombre determinado a través de más de una ruta. Dado que se puede aplicar un control de acceso diferente en estas rutas de acceso diferentes, el compilador elige la ruta de acceso que proporciona el mejor acceso. Consulte la siguiente figura.



Acceso mediante rutas de acceso de un gráfico de herencia

En la ilustración, se accede a un nombre declarado en la clase `VBase` siempre a través de la clase `RightPath`. El camino correcto es más accesible porque `RightPath` declara `VBase` como una `public` clase base, mientras que `LeftPath` declara `VBase` como `private`.

Vea también

[Referencia del lenguaje C++](#)

friend (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 6 minutos

En algunas circunstancias, es útil que una clase conceda acceso de nivel de miembro a funciones que no son miembros de la clase o a todos los miembros de una clase independiente. Estas funciones y clases gratuitas se conocen como *friends*, marcados por la palabra clave `friend`. Solo el implementador de la clase puede declarar cuáles son sus funciones o clases friend. Las funciones o clases no pueden hacerlo por sí mismas. En una definición de clase, use la palabra clave `friend` y el nombre de una función no miembro u otra clase para conceder acceso a los miembros privados y protegidos de la clase. En una definición de plantilla, un parámetro de tipo se puede declarar como `friend`.

Sintaxis

`friend-declaration:`

```
friend function-declaration  
friend function-definition  
friend elaborated-type-specifier ;  
friend simple-type-specifier ;  
friend typename-specifier ;
```

Declaraciones `friend`

Si declara una función `friend` que no se declaró previamente, esa función se exporta al ámbito de inclusión que no es de clase.

Las funciones declaradas en una declaración `friend` se tratan como si se hubieran declarado mediante la palabra clave `extern`. Para más información, consulte `extern`.

Aunque las funciones con ámbito global se pueden declarar como `friend` antes que los prototipos, las funciones miembro no se pueden declarar como `friend` antes de que aparezca la declaración de clase completa. En el código siguiente se muestra cómo se produce un error en esta declaración:

C++

```
class ForwardDeclared; // Class name is known.  
class HasFriends  
{
```

```
    friend int ForwardDeclared::IsAFriend(); // C2039 error expected
};
```

El ejemplo anterior introduce el nombre de clase `ForwardDeclared` en el ámbito, pero la declaración completa (específicamente, la parte que declara la función `IsAFriend`) no se conoce. Por consiguiente, la declaración de `friend` en la clase `HasFriends` genera un error.

En C++11, hay dos formas de declaraciones "friend" para una clase:

C++

```
friend class F;
friend F;
```

El primer formulario introduce una nueva clase `F` si no se encontró ninguna clase existente por ese nombre en el espacio de nombres más interno. C++11: el segundo formulario no presenta una nueva clase; se puede usar cuando la clase ya se ha declarado y debe usarse al declarar un parámetro de tipo de plantilla `typedef` como un `friend`.

Use `friend class F` cuando el tipo al que se hace referencia aún no se haya declarado:

C++

```
namespace NS
{
    class M
    {
        friend class F; // Introduces F but doesn't define it
    };
}
```

Se produce un error si usa `friend` con un tipo de clase que no se ha declarado:

C++

```
namespace NS
{
    class M
    {
        friend F; // error C2433: 'NS::F': 'friend' not permitted on data
declarations
    };
}
```

En el ejemplo siguiente, `friend F` hace referencia a la clase `F` que se declara fuera del ámbito de NS.

C++

```
class F {};
namespace NS
{
    class M
    {
        friend F; // OK
    };
}
```

Use `friend F` para declarar un parámetro de plantilla como friend:

C++

```
template <typename T>
class my_class
{
    friend T;
    //...
};
```

Use `friend F` para declarar una definición de tipo como friend:

C++

```
class Foo {};
typedef Foo F;

class G
{
    friend F; // OK
    friend class F // Error C2371 -- redefinition
};
```

Para declarar dos clases que son de tipo friend entre sí, la segunda clase completa se debe especificar como friend de la primera clase. La razón de esta restricción se debe a que el compilador solo tiene información suficiente para declarar funciones friend individuales en el punto donde se declara la segunda clase.

ⓘ Nota

Aunque la segunda clase completa debe ser definirse como friend en la primera clase, puede seleccionar las funciones de la primera clase que se definen como friend para la segunda clase.

funciones de confianza

Una función `friend` es una función que no es miembro de una clase pero tiene acceso a los miembros privados y protegidos de la clase. Las funciones friend no se consideran miembros de clase; son funciones externas normales que tienen privilegios de acceso especiales. No están en el ámbito de la clase y no se las llama usando los operadores de selección de miembro (`. y >`) a menos que sean miembros de otra clase. Una función `friend` la declara la clase que concede el acceso. La declaración `friend` se puede colocar en cualquier lugar de la declaración de clase. No se ve afectada por las palabras clave de control de acceso.

En el ejemplo siguiente se muestra una clase `Point` y una función friend, `ChangePrivate`. La función `friend` tiene acceso al miembro de datos privado del objeto `Point` que recibe como parámetro.

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  
}
```

Miembros de clase como friend

Las funciones miembro de clase se pueden declarar como de confianza en otras clases.

Considere el ejemplo siguiente:

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
```

En el ejemplo anterior, solo se concede a la función `A::Func1(B&)` acceso `friend` a la clase `B`. Por consiguiente, el acceso al miembro privado `_b` es correcto en `Func1` de la clase `A` pero no en `Func2`.

Una clase `friend` es una clase todas cuyas funciones miembro con funciones `friend` de una clase, es decir, cuyas funciones miembro tienen acceso a los miembros privados y protegidos de la otra clase. Suponga que la declaración `friend` de la clase `B` hubiera sido:

C++

```
friend class A;
```

En ese caso, a todas las funciones miembro de la clase `A` se les habría concedido acceso `friend` a la clase `B`. El código siguiente es un ejemplo de una clase `friend`:

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;}
};

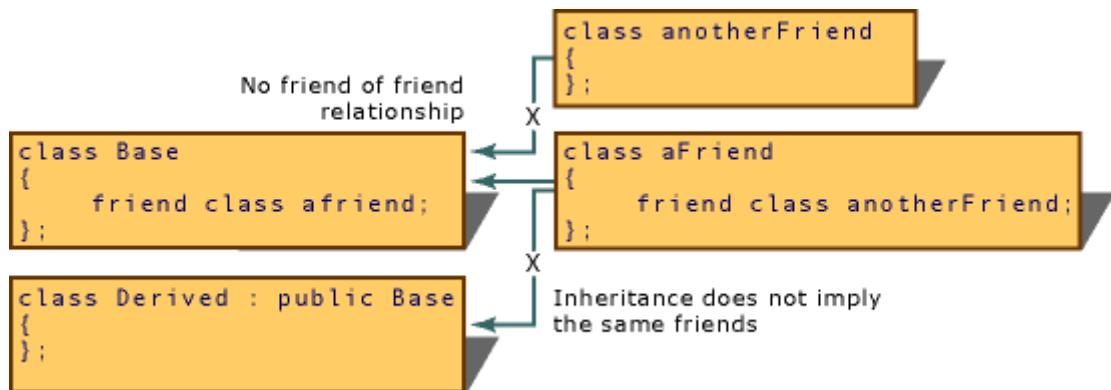
int main() {
    YourClass yc1;
    YourOtherClass yoc1;
    yc1.printMember();
    yoc1.change( yc1, 5 );
    yc1.printMember();
}
```

La declaración "friend" no es mutua a menos que se especifique explícitamente como tal. En el ejemplo anterior, las funciones miembro de `YourClass` no pueden tener acceso a los miembros privados de `YourOtherClass`.

Un tipo administrado (en C++/CLI) no puede tener funciones `friend`, clases `friend` ni interfaces `friend`.

La declaración "friend" no se hereda, lo que significa que las clases derivadas de `YourOtherClass` no pueden tener acceso a los miembros privados de `YourClass`. La declaración "friend" no es transitiva, por lo que las clases friend de `YourOtherClass` no pueden tener acceso a los miembros privados de `YourClass`.

La ilustración siguiente muestra cuatro declaraciones de clase: `Base`, `Derived`, `aFriend` y `anotherFriend`. Solo la clase `aFriend` tiene acceso directo a los miembros privados de `Base` (y a cualquier miembro `Base` que pueda haber heredado).



Implicaciones de relaciones de confianza

Definición `friend` insertadas

Las funciones friend se pueden definir (dado un cuerpo de función) dentro de las declaraciones de clase. Estas funciones son funciones insertadas. Como funciones insertadas de miembro, se comportan como si se hubieran definido inmediatamente después de haberse considerado todos los miembros de clase pero antes de cerrarse el ámbito de clase (el final de la declaración de clase). Las funciones friend definidas dentro de declaraciones de clase están en el ámbito de la clase envolvente.

Consulte también

[Palabras clave](#)

private (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Sintaxis

```
private:  
    [member-list]  
private base-class
```

Comentarios

Cuando precede a una lista de miembros de clase, la palabra clave `private` especifica que esos miembros son accesibles solo desde funciones miembro y friend de la clase. Esto se aplica a todos los miembros declarados hasta el especificador de acceso siguiente o el final de la clase.

Cuando precede al nombre de una clase base, la palabra clave `private` especifica que los miembros públicos y protegidos de la clase base son miembros privados de la clase derivada.

El acceso predeterminado de miembros de una clase es privado. El acceso predeterminado de miembros de una estructura o unión es público.

El acceso predeterminado de una clase base es privado para las clases y público para las estructuras. Las uniones no pueden tener clases base.

Para obtener información relacionada, consulte [friend](#), [public](#), [protected](#) y la tabla de acceso a miembros en [Controlar el acceso a los miembros de clase](#).

Específicos de /clr

En los tipos de CLR, las palabras clave del especificador de acceso de C++ (`public`, `private` y `protected`) pueden afectar a la visibilidad de los tipos y los métodos con respecto a los ensamblados. Para obtener más información, consulte [Control de acceso a miembros](#).

 Nota

Los archivos compilados con /LN no se ven afectados por este comportamiento. En este caso, todas las clases administradas (ya sean públicas o privadas) estarán visibles.

Específicos de END /clr

Ejemplo

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
}
```

Consulte también

Controlar el acceso a los miembros de clase

Palabras clave

protected (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Sintaxis

```
protected:  
    [member-list]  
protected base-class
```

Comentarios

La palabra clave `protected` especifica el acceso a los miembros de clase en *member-list* hasta el especificador de acceso siguiente (`public` o `private`) o el final de la definición de clase. Los miembros de clase declarados como `protected` solo los pueden usar los elementos siguientes:

- Funciones miembro de la clase que declaró originalmente estos miembros.
- Objetos friend de la clase que declaró originalmente estos miembros.
- Clases derivadas con acceso público o protegido desde la clase que declaró originalmente estos miembros.
- Clases directas derivadas de forma privada que también tienen acceso privado a miembros protegidos.

Cuando precede al nombre de una clase base, la palabra clave `protected` especifica que los miembros públicos y protegidos de la clase base son miembros protegidos de sus clases derivadas.

Los miembros protegidos no son tan privados como los miembros `private`, que solo son accesibles a los miembros de la clase en la que se declaran, pero no son tan públicos como los miembros `public`, que son accesibles en cualquier función.

Los miembros protegidos que también se declaran como `static` son accesibles a cualquier función miembro o friend de una clase derivada. Los miembros protegidos que no se declaran como `static` son accesibles a las funciones miembro y friend de una clase derivada solo mediante un objeto de la clase derivada, un puntero a esta o una referencia a esta.

Para obtener información relacionada, consulte [friend](#), [public](#), [private](#) y la tabla de acceso a miembros en [Control de acceso a miembros de clase](#).

Específicos de /clr

En los tipos de CLR, las palabras clave del especificador de acceso de C++ ([public](#), [private](#) y [protected](#)) pueden afectar a la visibilidad de los tipos y los métodos con respecto a los ensamblados. Para obtener más información, consulte [Control de acceso a miembros](#).

ⓘ Nota

Los archivos compilados con [/LN](#) no se ven afectados por este comportamiento. En este caso, todas las clases administradas (ya sean públicas o privadas) estarán visibles.

Específicos de END /clr

Ejemplo

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 derived class
}
```

Consulte también

[Controlar el acceso a los miembros de clase](#)

[Palabras clave](#)

public (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Sintaxis

```
public:  
    [member-list]  
public base-class
```

Comentarios

Cuando precede a una lista de miembros de clase, la palabra clave `public` especifica que esos miembros son accesibles desde cualquier función. Esto se aplica a todos los miembros declarados hasta el especificador de acceso siguiente o el final de la clase.

Cuando precede al nombre de una clase base, la palabra clave `public` especifica que los miembros públicos y protegidos de la clase base son miembros públicos y protegidos, respectivamente, de la clase derivada.

El acceso predeterminado de miembros de una clase es privado. El acceso predeterminado de miembros de una estructura o unión es público.

El acceso predeterminado de una clase base es privado para las clases y público para las estructuras. Las uniones no pueden tener clases base.

Para obtener más información, consulte [private](#), [protected](#), [friend](#) y la tabla de acceso a miembros en [Controlar el acceso a miembros de clase](#).

Específicos de /clr

En los tipos de CLR, las palabras clave del especificador de acceso de C++ (`public`, `private` y `protected`) pueden afectar a la visibilidad de los tipos y los métodos con respecto a los ensamblados. Para obtener más información, consulte [Control de acceso a miembros](#).

 Nota

Los archivos compilados con /LN no se ven afectados por este comportamiento. En este caso, todas las clases administradas (ya sean públicas o privadas) estarán visibles.

Específicos de END /clr

Ejemplo

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
}
```

Consulte también

[Controlar el acceso a los miembros de clase](#)

[Palabras clave](#)

Inicialización de llaves

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

No siempre es necesario definir un constructor para una `class`, especialmente si son relativamente sencillas. Los usuarios pueden inicializar objetos de una `class` o `struct` usando la inicialización uniforme, tal y como se muestra en el siguiente ejemplo:

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;
}

```

Cuando una `class` o `struct` no tiene constructor, proporcione los elementos de lista en el orden en que se declaran los miembros en `class`. Si `class` tiene un constructor, proporcione los elementos en el orden de los parámetros. Si un tipo tiene un constructor predeterminado, ya esté declarado de forma implícita o explícita, puede utilizar la inicialización de llave con las llaves vacías para invocarlo. Por ejemplo, la clase `class` siguiente se puede inicializar mediante la inicialización de llaves vacía y la no vacía:

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);
}

```

Si una clase tiene constructores no predeterminados, el orden en que los miembros de clase aparecen en el inicializador de llave es aquel en que aparecen los parámetros correspondientes en el constructor, no el orden en que se declaran los miembros (como ocurre con `class_a` en el ejemplo anterior). De lo contrario, si el tipo no tiene ningún constructor declarado, los inicializadores de miembro deben aparecer en el inicializador de llave en el mismo orden en que se declaran. En este caso, puede inicializar tantos

miembros públicos como desee, pero no puede omitir ningún miembro. En el ejemplo siguiente se muestra el orden que se utiliza en la inicialización de llave cuando no hay un constructor declarado:

```
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  
}
```

Si el constructor predeterminado se declara explícitamente pero se marca como eliminado, no se puede utilizar la inicialización de llave vacía:

```
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  
}
```

Puede utilizar la inicialización de llave en cualquier parte donde realizaría normalmente la inicialización, por ejemplo, como un parámetro de función o un valor devuelto, o con la palabra clave `new`:

```
C++
```

```
class_d* cf = new class_d{4.5};  
kr->add_d({ 4.5 });  
return { 4.5 };
```

En el modo `/std:c++17` y en versiones posteriores, las reglas para la inicialización de llaves vacías son ligeramente más restrictivas. Consulte [Constructores derivados e inicialización de agregado extendida](#).

Constructores initializer_list

`initializer_list Class` representa una lista de objetos de un tipo especificado que se puede utilizar en un constructor y en otros contextos. Puede construir `initializer_list` mediante la inicialización de llave:

C++

```
initializer_list<int> int_list{5, 6, 7};
```

ⓘ Importante

Para utilizar esta clase, debe incluir el encabezado `<initializer_list>`.

`initializer_list` puede copiarse. En este caso, los miembros de la nueva lista son referencias a los miembros de la lista original:

C++

```
initializer_list<int> ilist1{ 5, 6, 7 };  
initializer_list<int> ilist2( ilist1 );  
if (ilist1.begin() == ilist2.begin())  
    cout << "yes" << endl; // expect "yes"
```

Las clases contenedoras de la biblioteca estándar y `string`, `wstring` y `regex`, tienen constructores `initializer_list`. En los ejemplos siguientes se muestra cómo realizar la inicialización de llave con estos constructores:

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' };
```

Consulte también

[Clases y structs](#)

[Constructores](#)

Duración de objetos y administración de recursos (RAII)

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

A diferencia de los lenguajes administrados, C++ no tiene *recolección automática de elementos no utilizados*, un proceso interno que libera la memoria del montón y otros recursos a medida que se ejecuta un programa. Un programa de C++ es responsable de devolver todos los recursos adquiridos al sistema operativo. Si no se libera un recurso sin usar, se denomina *filtración*. Los recursos filtrados no están disponibles para otros programas hasta que se cierre el proceso. Las fugas de memoria en particular son una causa común de errores en la programación de estilo C.

El C++ moderno evita el uso de la memoria del montón tanto como sea posible mediante la declaración de objetos en la pila. Cuando un recurso es demasiado grande para la pila, debe ser *propiedad* de un objeto. A medida que se inicializa el objeto, adquiere el recurso que posee. A continuación, el objeto es responsable de liberar el recurso en su destructor. El propio objeto propietario se declara en la pila. El principio de que *los objetos poseen recursos* también se conocen como "la adquisición de recursos es la inicialización" o RAII.

Cuando un objeto de pila propietario de recursos sale del ámbito, se invoca automáticamente su destructor. De este modo, la recolección de elementos no utilizados en C++ está estrechamente relacionada con la duración del objeto y es determinista. Un recurso siempre se libera en un punto conocido del programa, que puede controlar. Solo los destructores deterministas como los de C++ pueden controlar igualmente los recursos de memoria y no memoria.

En el siguiente ejemplo se muestra un objeto simple `w`. Se declara en la pila en el ámbito de la función y se destruye al final del bloque de función. El objeto `w` no posee *ningún recurso* (por ejemplo, memoria asignada por montón). Su único miembro `g` se declara en la pila y simplemente sale del ámbito junto con `w`. No se necesita ningún código especial en el destructor `widget`.

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(); }"

```

En el ejemplo siguiente, `w` posee un recurso de memoria y, por tanto, debe tener código en su destructor para eliminar la memoria.

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

```

Desde C++11, hay una mejor manera de escribir el ejemplo anterior: mediante un puntero inteligente de la biblioteca estándar. Un puntero inteligente controla la asignación y la eliminación de la memoria de la que es propietario. El uso de un puntero inteligente elimina la necesidad de un destructor explícito en la clase `widget`.

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
```

Mediante el uso de punteros inteligentes para la asignación de memoria, puede eliminar la posibilidad de pérdidas de memoria. Este modelo funciona para otros recursos, como identificadores de archivo o sockets. Puede administrar sus propios recursos de forma similar en las clases. Para obtener más información, consulte [Punteros inteligentes](#).

El diseño de C++ garantiza que los objetos se destruyen cuando salen del ámbito. Es decir, se destruyen a medida que salen bloques, en orden inverso de construcción. Cuando se destruye un objeto, sus bases y miembros se destruyen en un orden determinado. Los objetos declarados fuera de cualquier bloque, en el ámbito global, pueden provocar problemas. Puede ser difícil depurarlo si el constructor de un objeto global produce una excepción.

Consulte también

[Aquí está otra vez C++](#)

[Referencia del lenguaje C++](#)

[Biblioteca estándar de C++](#)

Pimpl para encapsulación en tiempo de compilación (C++ moderno)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

La expresión *pimpl* es una técnica moderna de C++ para ocultar la implementación, minimizar el acoplamiento y separar interfaces. Pimpl es la abreviatura de "puntero a la implementación". Es posible que ya esté familiarizado con el concepto, pero que lo conozca por otros nombres como expresión Cheshire Cat o Compiler Firewall.

¿Por qué usar pimpl?

Así es como la expresión *pimpl* puede mejorar el ciclo de vida de desarrollo de software:

- Minimización de dependencias de compilación.
- Separación de la interfaz y la implementación.
- Portabilidad.

Encabezado de pimpl

```
C++

// my_class.h
class my_class {
    // ... all public and protected stuff goes here ...
private:
    class impl; unique_ptr<impl> pimpl; // opaque type here
};
```

La expresión *pimpl* evita las cascadas de recompilación y los diseños de objetos frágiles. Es adecuado para tipos conocidos (transitivamente).

Implementación de pimpl

Defina la clase `impl` en el archivo .cpp.

```
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 ...
}
```

Procedimientos recomendados

Considere si debe agregar compatibilidad con la especialización de intercambio que no produce excepciones.

Consulte también

[Aquí está otra vez C++](#)

[Referencia del lenguaje C++](#)

[Biblioteca estándar de C++](#)

Portabilidad en los límites de ABI

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Use tipos y convenciones suficientemente portátiles en los límites de la interfaz binaria. Un "tipo portátil" es un tipo integrado de C o una estructura que contiene solo tipos integrados de C. Los tipos de clases solo se pueden usar cuando el autor y el destinatario de la llamada tienen el mismo diseño, usan la misma convención de llamada, etc. Eso solo es posible cuando se han usado el mismo compilador y la misma configuración de compilador para realizar la compilación.

Cómo acoplar una clase para la portabilidad de C

Cuando los autores de llamadas se pueden compilar con otro compilador o lenguaje, "acople" a una API externa de "C" con una convención de llamada específica:

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"
}
```

Consulte también

[Aquí está otra vez C++](#)

[Referencia del lenguaje C++](#)

[Biblioteca estándar de C++](#)

Constructores (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 19 minutos

Para personalizar cómo inicializa una clase sus miembros o para invocar funciones cuando se crea un objeto de la clase, defina un *constructor*. Un constructor tiene el mismo nombre que la clase y no devuelve ningún valor. Puede definir tantos constructores sobrecargados como sea necesario para personalizar la inicialización de varias maneras. Normalmente, los constructores tienen accesibilidad pública para que el código que no pertenece a la jerarquía de herencia ni a la definición de la clase pueda crear objetos de la clase. Pero también puede declarar un constructor como `protected` o `private`.

Opcionalmente, los constructores pueden tomar una lista de inicializadores de miembros. Es una manera más eficaz de inicializar los miembros de la clase que asignar valores en el cuerpo del constructor. En el ejemplo siguiente, se muestra una clase `Box` con tres constructores sobrecargados. Los dos últimos utilizan listas de inicializadores de miembros:

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 };
};
```

Al declarar una instancia de una clase, el compilador elige qué constructor invocar en función de las reglas de resolución de sobrecargas:

```
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)  
}
```

- Los constructores se pueden declarar como `inline`, `explicit`, `friend` o `constexpr`.
- Un constructor puede inicializar un objeto que se ha declarado como `const`, `volatile` o `const volatile`. El objeto se convierte en `const` después de que finalice el constructor.
- Para definir un constructor en un archivo de implementación, asígnele un nombre completo como a cualquier otra función miembro: `Box::Box(){...}`.

Listas de inicializadores de miembros

Opcionalmente, un constructor puede tener una *lista de inicializadores de miembros*, que inicializa los miembros de la clase antes de que se ejecute el cuerpo del constructor. (Una lista de inicializadores de miembros no es lo mismo que una *lista de inicializadores* de tipo `std::initializer_list<T>`).

Son preferibles las listas de inicializadores de miembros antes que la asignación de valores en el cuerpo del constructor. Una lista de inicializadores de miembros inicializa directamente los miembros. En el ejemplo siguiente, se muestra la lista de inicializadores de miembros, que consta de todas las expresiones `identifier(argument)` después de los dos puntos:

```
C++  
  
Box(int width, int length, int height)  
    : m_width(width), m_length(length), m_height(height)  
{}
```

El identificador debe hacer referencia a un miembro de la clase; se inicializa con el valor del argumento. El argumento puede ser uno de los parámetros del constructor, una llamada de función o `std::initializer_list<T>`.

Los miembros `const` y los miembros del tipo de referencia se deben inicializar en la lista de inicializadores de miembros.

Para asegurarse de que las clases base se hayan inicializado completamente antes de que se ejecute el constructor derivado, llame a cualquier constructor de clase base con parámetros en la lista de inicializadores.

Constructores predeterminados

Los *constructores predeterminados* normalmente no tienen parámetros, pero pueden tener parámetros con valores predeterminados.

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){}
...
}
```

Los constructores predeterminados son una de las [funciones miembro especiales](#). Si no se declara ningún constructor en una clase, el compilador proporciona un constructor predeterminado `inline` implícito.

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
}
```

Si se basa en un constructor predeterminado implícito, asegúrese de inicializar los miembros en la definición de la clase, como se muestra en el ejemplo anterior. Sin esos inicializadores, los miembros no se inicializarían y la llamada a Volume() produciría un valor de elemento no utilizado. En general, se recomienda inicializar los miembros de esta manera incluso cuando no se basan en un constructor predeterminado implícito.

Puede impedir que el compilador genere un constructor predeterminado implícito si lo define como [eliminado](#):

C++

```
// Default constructor
Box() = delete;
```

Un constructor predeterminado generado por el compilador se definirá como eliminado si algún miembro de la clase no se puede construir de manera predeterminada. Por ejemplo, todos los miembros del tipo de clase y sus miembros de tipo de clase deben tener un constructor predeterminado y destructores a los que se pueda acceder. Todos los miembros de datos del tipo de referencia y todos los miembros [const](#) deben tener un inicializador de miembros predeterminado.

Al llamar a un constructor predeterminado generado por el compilador e intentar usar paréntesis, se emite una advertencia:

C++

```
class myclass{};
int main(){
myclass mc();    // warning C4930: prototyped function not called (was a
variable definition intended?)
}
```

Esta instrucción es un ejemplo del problema "Análisis más acuciante". Puede interpretar `myclass md();` como una declaración de función o como la invocación de un constructor predeterminado. Dado que los analizadores de C++ favorecen las declaraciones sobre otras cosas, la expresión se trata como una declaración de función. Para obtener más información, consulte [Análisis más acuciante ↗](#).

Si se declaran constructores no predeterminados, el compilador no proporcionará un constructor predeterminado:

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  
}
```

Si una clase no tiene ningún constructor predeterminado, no se puede crear una matriz de objetos de esa clase únicamente mediante una sintaxis de corchetes. Por ejemplo, dado el bloque de código anterior, no se puede declarar una matriz de objetos Box de esta manera:

C++

```
Box boxes[3]; // C2512: no appropriate default constructor available
```

Sin embargo, puede utilizar un conjunto de listas de inicializadores para inicializar una matriz de objetos Box:

C++

```
Box boxes[3]{ { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

Para obtener más información, consulte [Inicializadores](#).

Constructores de copias

Un *constructor de copia* inicializa un objeto copiando los valores de los miembros de un objeto del mismo tipo. Si los miembros de la clase son todos tipos simples, como valores escalares, el constructor de copia generado por el compilador es suficiente y no es necesario definir el suyo propio. Si la clase requiere una inicialización más compleja, debe implementar un constructor de copia personalizado. Por ejemplo, si un miembro

de clase es un puntero, debe definir un constructor de copia para asignar nueva memoria y copiar los valores del objeto al que apunta el otro. El constructor de copia generado por el compilador simplemente copia el puntero, por lo que el nuevo puntero sigue apuntando a la ubicación de memoria del otro.

Un constructor de copia puede tener una de estas firmas:

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");
```

Al definir un constructor de copia, también debe definir un operador de asignación de copia (=). Para obtener más información, consulte [Asignación y Constructores de copia y operadores de asignación de copia \(C++\)](#).

Puede evitar que el objeto se copie definiendo el constructor de copia como eliminado:

C++

```
Box (const Box& other) = delete;
```

Al intentar copiar el objeto, se produce el error *C2280: se intenta hacer referencia a una función eliminada*.

Constructores de movimiento

Un *constructor de movimiento* es una función miembro especial que mueve la propiedad de los datos de un objeto existente a una nueva variable sin copiar los datos originales. Toma una referencia rvalue como primer parámetro y los parámetros posteriores deben tener valores predeterminados. Los constructores de movimiento pueden aumentar significativamente la eficacia del programa al pasar objetos grandes.

C++

```
Box(Box&& other);
```

El compilador elige un constructor de movimiento cuando otro objeto del mismo tipo inicializa el objeto, si el otro objeto está a punto de ser destruido y ya no necesita sus

recursos. En el ejemplo siguiente, se muestra un caso cuando se selecciona un constructor de movimiento mediante la resolución de sobrecarga. En el constructor que llama a `get_Box()`, el valor devuelto es un valor *xvalue* (valor que va a expirar). No está asignado a ninguna variable y, por lo tanto, está a punto de salir del ámbito. Para proporcionar motivación para este ejemplo, vamos a proporcionar a `Box` un vector grande de cadenas que representan su contenido. En lugar de copiar el vector y sus cadenas, el constructor de movimiento lo "roba" del valor "box" que va a expirar para que el vector ahora pertenezca al nuevo objeto. La llamada a `std::move` es todo lo que se necesita porque las clases `vector` y `string` implementan sus propios constructores de movimiento.

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;
}

```

Si una clase no define un constructor de movimiento, el compilador genera uno implícito si no hay ningún constructor de copia declarado por el usuario, operador de asignación de copia, operador de asignación de movimiento ni destructor. Si no se define ningún constructor de movimiento explícito o implícito, las operaciones que usarían de otro modo un constructor de movimiento usan el constructor de copia en su lugar. Si una clase declara un constructor de movimiento o un operador de asignación de movimiento, el constructor de copia declarado implícitamente se define como eliminado.

Un constructor de movimiento declarado implícitamente se define como eliminado si alguno de los miembros que son tipos de clase carece de un destructor o si el compilador no puede determinar qué constructor usar para la operación de movimiento.

Para obtener más información sobre cómo escribir un constructor de movimiento no trivial, consulte [Constructores de movimiento y operadores de asignación de movimiento \(C++\)](#).

Constructores explícitamente establecidos como predeterminados y eliminados

Puede establecer explícitamente constructores de copia *predeterminados*, constructores predeterminados, constructores de movimiento, operadores de asignación de copia, operadores de asignación de movimiento y destructores. Puede *eliminar* explícitamente todas las funciones miembro especiales.

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;
    //...
};
```

Para obtener más información, consulte [Funciones establecidas como valor predeterminado y eliminadas explícitamente](#).

Constructores `constexpr`

Un constructor se puede declarar como `constexpr` si:

- Se declara como predeterminado o satisface todas las condiciones de las [funciones `constexpr`](#) en general.
- La clase no tiene clases base virtuales.
- Cada uno de los parámetros es un tipo [literal](#).
- El cuerpo no es una función try-block.
- Se han inicializado todos los miembros de datos no estáticos y los subobjetos de clase base.
- Si la clase es (a) una unión que tiene miembros variantes o (b) tiene uniones anónimas, solo se inicializa uno de los miembros de la unión.
- Todos los miembros de datos no estáticos del tipo de clase y todos los subobjetos de clase base tienen un constructor `constexpr`.

Constructores de lista de inicializadores

Si un constructor toma `std::initializer_list<T>` como parámetro y cualquier otro parámetro tiene argumentos predeterminados, se selecciona ese constructor en la resolución de sobrecarga cuando se crea una instancia de la clase mediante la inicialización directa. Puede usar el elemento `initializer_list` para inicializar cualquier miembro que pueda aceptarlo. Por ejemplo, supongamos que la clase `Box` (mostrada anteriormente) tiene un miembro `m_contents` de tipo `std::vector<string>`. Puede proporcionar un constructor de esta manera:

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)
{}
```

Y, a continuación, crear objetos `Box` de esta forma:

C++

```
Box b{ "apples", "oranges", "pears" }; // or ...
Box b2(initializer_list<string> { "bread", "cheese", "wine" }, 2, 4, 6);
```

Constructores explícitos

Si una clase tiene un constructor con un solo parámetro, o si todos los parámetros excepto uno tienen un valor predeterminado, el tipo de parámetro se puede convertir implícitamente en el tipo de clase. Por ejemplo, si la clase `Box` tiene un constructor como este:

C++

```
Box(int size): m_width(size), m_length(size), m_height(size){}
```

Se puede inicializar un objeto `Box` de esta manera:

C++

```
Box b = 42;
```

O pasar un valor `int` a una función que toma un objeto `Box`:

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);
```

Estas conversiones pueden ser útiles en algunos casos, pero lo más habitual es que provoquen errores sutiles, pero graves, en el código. Como regla general, es conveniente usar la palabra clave `explicit` en un constructor (y los operadores definidos por el usuario) para evitar esta clase de conversión de tipos implícita:

C++

```
explicit Box(int size): m_width(size), m_length(size), m_height(size){}
```

Cuando el constructor es explícito, esta línea provoca un error del compilador:
`ShippingOrder so(42, 10.8);`. Para obtener más información, consulte [Conversiones de tipo definidas por el usuario \(C++\)](#).

Orden de construcción

Un constructor realiza su trabajo en este orden:

1. Llama a los constructores miembros y de clase base en el orden en que se declararon.
2. Si la clase se deriva de clases base virtuales, inicializa los punteros base virtuales del objeto.
3. Si la clase tiene o hereda funciones virtuales, inicializa los punteros de funciones virtuales del objeto. Los punteros de funciones virtuales apuntan a la tabla de funciones virtuales de la clase para permitir el enlace correcto de las llamadas de funciones virtuales al código.
4. Ejecuta cualquier código en el cuerpo de su función.

En el ejemplo siguiente se muestra el orden en el que los constructores miembros y de clase base se llaman en el constructor para una clase derivada. En primer lugar, se llama

al constructor base. A continuación, se inicializan los miembros de clase base en el orden en el que aparecen en la declaración de clase. Por último, se llama al constructor derivado.

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;
```

Este es el resultado:

Output

```
Contained1 ctor
Contained2 ctor
BaseContainer ctor
```

```
Contained3 ctor  
DerivedContainer ctor
```

Un constructor de clase derivada siempre llama a un constructor de clase base, por lo que se pueden usar clases base completamente construidas antes de realizar cualquier trabajo adicional. Los constructores de clase base se llaman en orden de derivación: por ejemplo, si `ClassA` se deriva de `ClassB`, que se deriva de `ClassC`, se llama primero al constructor de `ClassC`, después al constructor de `ClassB` y, por último, al constructor de `ClassA`.

Si una clase base no tiene un constructor predeterminado, debe proporcionar los parámetros del constructor de la clase base en el constructor de la clase derivada:

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);  
}
```

Si un constructor inicia una excepción, el orden de destrucción es el inverso al orden de la construcción:

1. El código del cuerpo de la función de constructor se desenredará.

2. Los objetos miembro y de la clase base se destruirán en el orden inverso de la declaración.
3. Si el constructor no delega, se destruirán todos los miembros y objetos de clase base totalmente construidos. Sin embargo, dado que el objeto en sí no está totalmente construido, el destructor no se ejecuta.

Constructores derivados e inicialización de agregado extendida

Si el constructor de una clase base no es público, pero es accesible para una clase derivada, no puede usar llaves vacías para inicializar un objeto del tipo derivado en modo `/std:c++17` y versiones posteriores en Visual Studio 2017 y versiones posteriores.

En el ejemplo siguiente se muestra el comportamiento correspondiente de C++14:

```
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.
```

En C++ 17, `Derived` ahora se considera un tipo de agregado. Eso significa que la inicialización de `Base` mediante el constructor privado predeterminado se produce directamente como parte de la regla de inicialización de agregados extendida. Anteriormente, se llamaba al constructor privado `Base` mediante el constructor `Derived` y se realizaba correctamente debido a la declaración `friend`.

En el ejemplo siguiente, se muestra el comportamiento de C++17 en Visual Studio 2017 y versiones posteriores en el modo `/std:c++17`:

```
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'

```

Constructores de clases que tienen herencia múltiple

Si una clase se deriva de varias clases base, los constructores de clase base se invocan en el orden en el que se enumeran en la declaración de la clase derivada:

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;
}

```

Debería esperar los siguientes resultados:

Output

```
BaseClass1 ctor  
BaseClass2 ctor  
BaseClass3 ctor  
DerivedClass ctor
```

Constructores de delegación

Un *constructor de delegación* llama a otro constructor de la misma clase para realizar algunas de las tareas de inicialización. Esta característica es útil cuando tiene varios constructores que todos tienen que realizar un trabajo similar. Puede escribir la lógica principal en un constructor e invocarla desde otros. En el siguiente ejemplo trivial, Box(int) delega su trabajo en Box(int,int,int):

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  
};
```

El objeto creado por los constructores se inicializa totalmente en cuanto finaliza cualquiera de los constructores. Para obtener más información, consulte [Constructores de delegación](#).

Constructores que heredan (C++11)

Una clase derivada puede heredar los constructores de una clase base directa mediante una declaración `using`, como se muestra en el ejemplo siguiente:

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 y versiones posteriores: la instrucción `using` en modo `/std:c++17` y versiones posteriores incluye en el ámbito a todos los constructores de la clase base, excepto los que tienen una firma idéntica a la de los constructores de la clase derivada. En general, es mejor usar constructores que heredan cuando la clase derivada no declara ningún constructor ni miembro de datos nuevo.

Una plantilla de clase puede heredar todos los constructores de un argumento de tipo si dicho tipo especifica una clase base:

C++

```
template< typename T >
class Derived : T {
    using T::T;    // declare the constructors from T
    // ...
};
```

Una clase derivada no puede heredar de varias clases base si esas clases base tienen constructores con una firma idéntica.

Constructores y clases compuestas

Las clases que contienen miembros de tipo de clase se conocen como *clases compuestas*. Cuando se crea un miembro de tipo de clase compuesta, se llama al constructor antes que al propio constructor de la clase. Si una clase contenida carece de un constructor predeterminado, debe utilizar una lista de inicializaciones en el constructor de la clase compuesta. En el ejemplo anterior de `StorageBox`, si cambia el tipo de la variable miembro `m_label` a una nueva clase `Label`, debe llamar al constructor de la clase base e inicializar la variable `m_label` en el constructor `StorageBox`:

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"});
}
```

En esta sección

- Constructores de copia y operadores de asignación de copia
- Constructores de movimiento y operadores de asignación de movimiento
- Constructores de delegación

Consulte también

[Clases y estructuras](#)

Constructores de copia y operadores de asignación de copia (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

ⓘ Nota

A partir de C++ 11, se admiten dos tipos de asignación en el lenguaje: *asignación de copia* y *asignación de movimiento*. En este artículo, "asignación" significa asignación de copia a menos que se establezca explícitamente lo contrario. Para obtener información sobre la asignación de movimiento, consulte [Operadores de constructores de movimiento y asignaciones de movimiento \(C++\)](#).

Tanto la operación de asignación como la operación de inicialización provocan que los objetos se copien.

- **Asignación:** cuando el valor de un objeto se asigna a otro objeto, el primer objeto se copia en el segundo objeto. Por lo tanto, este código copia el valor de `b` en `a`:

C++

```
Point a, b;  
...  
a = b;
```

- **Inicialización:** la inicialización se produce cuando se declara un nuevo objeto, cuando se pasan argumentos de función por valor o cuando se devuelve por valor de una función.

Puede definir la semántica de "copy" para los objetos de tipo de clase. Por ejemplo, considere este código:

C++

```
TextFile a, b;  
a.Open( "FILE1.DAT" );  
b.Open( "FILE2.DAT" );  
b = a;
```

El código anterior podría significar "copiar el contenido de FILE1.DAT en FILE2.DAT" o podría significar "omitar FILE2.DAT y convertir `b` en un segundo identificador para

FILE1.DAT". Debe adjuntar la semántica de copia adecuada a cada clase, como se muestra a continuación:

- Use un operador de asignación `operator=` que devuelva una referencia al tipo de clase y tome un parámetro que se pasa por referencia `const`; por ejemplo
`ClassName& operator=(const ClassName& x);`.
- Uso del constructor de copias.

Si no declara un constructor de copias, el compilador genera uno automáticamente miembro a miembro. Del mismo modo, si no declara una asignación de copia, el compilador genera una automáticamente miembro a miembro. Declarar un constructor de copias no suprime el operador de asignación de copias generado por el compilador, ni viceversa. Si implementa cualquiera de ellos, se recomienda que implemente también el otro. Cuando se implementan ambos, el significado del código es claro.

El constructor de copias toma un argumento de tipo `ClassName&`, donde `ClassName` es el nombre de la clase. Por ejemplo:

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 Nota

Haga que el tipo del argumento del constructor de copias `const ClassName&` sea siempre que sea posible. Esto evita que el constructor de copias modifique accidentalmente el objeto copiado. También permite copiar objetos `const`.

Constructores de copias generados por el compilador

Los constructores de copia generados por el compilador, como los constructores de copia definidos por el usuario, tienen un único argumento de tipo "referencia a *className*". Una excepción es cuando todas las clases base y las clases miembro tienen constructores de copia declarados como tomar un único argumento de tipo `const className&`. En ese caso, el argumento del constructor de copias generado por el compilador también es `const`.

Cuando el tipo de argumento al constructor de copias no es `const`, la inicialización que se realiza copiando un objeto `const` genera un error. Lo contrario no es cierto: si el argumento es `const`, puede inicializar copiando un objeto que no sea `const`.

Los operadores de asignación generados por el compilador siguen el mismo patrón para `const`. Toman un solo argumento de tipo `ClassName&` a menos que los operadores de asignaciones de todas las clases base y clases de miembro tomen argumentos de tipo `const ClassName&`. En este caso, el operador de asignación generado para la clase toma un argumento `const`.

ⓘ Nota

Cuando los constructores de copias, generados por el compilador o definidos por el usuario, inicializan las clases base virtuales, estas se inicializan solo una vez: en el momento en que se construyen.

Las implicaciones son similares a las del constructor de copias. Cuando el tipo del argumento no es `const`, la asignación de un objeto `const` genera un error. Lo contrario no es cierto: si un valor `const` se asigna a un valor que no es `const`, la asignación se realiza correctamente.

Para obtener más información sobre los operadores de asignaciones sobrecargados, vea [Asignación](#).

Constructores de movimiento y operadores de asignación de movimiento (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 6 minutos

En este tema se describe cómo escribir un *constructor de movimiento* y un operador de asignaciones de movimiento para una clase de C++. Un constructor de movimiento permite que los recursos que pertenecen a un objeto rvalue se muevan a un lvalue sin copiar. Para obtener más información sobre la semántica de movimiento, consulte el [Declarador de referencia de Rvalue: &&](#).

Este tema se basa en la siguiente clase de C++, `MemoryBlock`, que administra un búfer de memoria.

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.
};

```

Los procedimientos siguientes describen cómo escribir un constructor de movimiento y un operador de asignación de movimiento para la clase de C++ de ejemplo.

Para crear un constructor de movimiento para una clase de C++

1. Defina un método de constructor vacío que tome una referencia de valor R al tipo de clase como su parámetro, como se muestra en el ejemplo siguiente:

C++

```
MemoryBlock(MemoryBlock&& other)
: _data(nullptr)
, _length(0)
{}
```

2. En el constructor de movimiento, asigne los miembros de datos de clase del objeto de origen al objeto que se está construyendo:

C++

```
_data = other._data;
_length = other._length;
```

3. Asigne los miembros de datos del objeto de origen a valores predeterminados.

Esto evita que el destructor libere varias veces los recursos (tales como la memoria):

C++

```
other._data = nullptr;
other._length = 0;
```

Para crear un operador de asignaciones de movimiento para una clase de C++

1. Defina un operador de asignación vacío que tome una referencia de valor R al tipo de clase como su parámetro y devuelva una referencia al tipo de clase, como se muestra en el ejemplo siguiente:

C++

```
MemoryBlock& operator=(MemoryBlock&& other)
{}
```

2. En el operador de asignación de movimiento, agregue una instrucción condicional que no realice ninguna operación si intenta asignar el objeto a sí mismo.

C++

```
if (this != &other)
{
}
```

3. En la instrucción condicional, libere los recursos (tales como la memoria) del objeto al que se asigna.

El ejemplo siguiente libera el miembro `_data` del objeto al que se asigna:

C++

```
// Free the existing resource.
delete[] _data;
```

Siga los pasos 2 y 3 del primer procedimiento para transferir los miembros de datos del objeto de origen al objeto que se construye:

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. Devuelva una referencia al objeto actual, como se muestra en el ejemplo siguiente:

C++

```
return *this;
```

Ejemplo: Operador de asignación y constructor de movimiento completo

En el ejemplo siguiente se muestra el constructor de movimiento y el operador de asignación de movimiento completos para la clase `MemoryBlock`:

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;
}

```

Ejemplo Uso de la semántica de movimiento para mejorar el rendimiento

El ejemplo siguiente muestra cómo la semántica de transferencia de recursos puede mejorar el rendimiento de las aplicaciones. El ejemplo agrega dos elementos a un objeto vectorial y después inserta un nuevo elemento entre los dos existentes. En la

clase `vector`, usa semántica de transferencia de recursos para realizar la operación de inserción eficazmente moviendo los elementos del vector en lugar de copiarlos.

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));
}
```

Este ejemplo produce el siguiente resultado:

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.
```

Antes de Visual Studio 2010, este ejemplo generó la siguiente salida:

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.
```

La versión de este ejemplo que usa semántica de movimiento de recursos es más eficiente que la versión que no la usa, porque realiza menos operaciones de copia, asignación de memoria y desasignación de memoria.

Programación sólida

Para evitar pérdidas de recursos, libere siempre los recursos (tales como memoria, identificadores de archivos y sockets) en el operador de asignación de movimiento.

Para evitar la destrucción irrecuperable de recursos, administre correctamente la autoasignación en el operador de asignación de movimiento.

Si proporciona tanto un constructor de movimiento como un operador de asignación de movimiento para la clase, puede eliminar código redundante escribiendo el constructor de movimiento para llamar al operador de asignación de movimiento. En el ejemplo siguiente se muestra una versión revisada del constructor de movimiento que llama al operador de asignación de movimiento:

C++

```
// Move constructor.
MemoryBlock(MemoryBlock&& other) noexcept
    : _data(nullptr)
    , _length(0)
{
    *this = std::move(other);
}
```

La función `std::move` convierte el valor lvalue `other` en rvalue.

Consulte también

[Declarador de referencias rvalue: &&](#)
[std::move](#)

Constructores de delegación

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Muchas clases tienen varios constructores que realizan acciones similares, por ejemplo, validan parámetros:

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;
    }
};
```

Puede reducir el código repetitivo si agrega una función que realice toda la validación, pero el código de `class_c` sería más fácil de entender y mantener si un constructor pudiera delegar alguna parte del trabajo en otro. Para agregar la delegación de constructores, utilice la sintaxis `constructor (. . .) : 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 };
}

```

A medida que recorra paso a paso el ejemplo anterior, observe que el constructor `class_c(int, int, int)` llama primero al constructor `class_c(int, int)`, que a su vez llama a `class_c(int)`. Cada uno de los constructores realiza solo el trabajo que no realizan los otros constructores.

El primer constructor al que se llama inicializa el objeto para que todos sus miembros se inicialicen en ese momento. No se puede realizar la inicialización de miembros en un constructor que delegue a otro constructor, como se muestra aquí:

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;
};

```

En el ejemplo siguiente se muestra el uso de los inicializadores de miembro de datos no estático. Observe que, si un constructor inicializa también un miembro de datos determinado, el inicializador de miembro se invalida:

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;
}
```

La sintaxis de delegación de constructores no impide la creación accidental de recursividad de constructores (el Constructor1 llama al Constructor2, que llama al Constructor1) y no se genera ningún error hasta que se produzca un desbordamiento de pila. Es responsabilidad del programador evitar los ciclos.

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() { }
};
```

Destructores (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 7 minutos

Un destructor es una función miembro que se invoca automáticamente cuando el objeto sale del ámbito o se destruye explícitamente mediante una llamada a `delete`. Un destructor tiene el mismo nombre que la clase precedido por una tilde (~). Por ejemplo, el destructor de la clase `String` se declara como: `~String()`.

Si no define un destructor, el compilador proporciona uno predeterminado; para muchas clases, esto es suficiente. Solo tiene que definir un destructor personalizado cuando la clase almacena los identificadores de los recursos del sistema que deben liberarse, o los punteros que poseen la memoria a la que apuntan.

Considere la siguiente declaración de una clase `String`:

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...");  
}
```

En el ejemplo anterior, el destructor `String::~String` usa el operador `delete` para desasignar el espacio asignado dinámicamente para el almacenamiento de texto.

Declaración de destructores

Los destructores son funciones con el mismo nombre que la clase pero precedidos por una tilde (~).

Varias reglas rigen la declaración de destructores. Destructores:

- No acepte argumentos.
- No devuelva un valor (o `void`).
- No se puede declarar como `const`, `volatile` o `static`. Aun así, se pueden invocar para la destrucción de objetos declarados como `const`, `volatile` o `static`.
- Se pueden declarar como `virtual`. Con destructores virtuales, puede destruir objetos sin conocer su tipo: el destructor correcto para el objeto se invoca mediante el mecanismo de función virtual. Los destructores también se pueden declarar como funciones virtuales puras para clases abstractas.

Usar destructores

Se llama a los destructores cuando se produce alguno de los eventos siguientes:

- Un objeto local (automático) con ámbito de bloque sale de ámbito.
- Un objeto asignado mediante el operador `new` se desasigna explícitamente con `delete`.
- La duración de un objeto temporal termina.
- Un programa termina y hay objetos globales o estáticos.
- Se llama explícitamente al destructor mediante el nombre completo de la función de destructor.

Los destructores pueden llamar libremente a funciones miembro de clase y acceder a datos de miembros de clase.

Hay dos restricciones en el uso de destructores:

- No puede tomar su dirección.
- Las clases derivadas no heredan el destructor de su clase base.

Orden de destrucción

Cuando un objeto sale del ámbito o se elimina, la secuencia de eventos para su completa destrucción es la siguiente:

1. Se llama al destructor de clase y se ejecuta el cuerpo de la función destructora.
2. Los destructores de los objetos miembro no estáticos se llaman en el orden inverso al que aparecen en la declaración de clase. La lista opcional de inicialización de miembros usada en la construcción de estos miembros no afecta al orden de construcción o destrucción.
3. Los destructores para las clases base no virtuales se llaman en el orden inverso al de la declaración.
4. Los destructores para las clases base virtuales se llaman en el orden inverso al de la declaración.

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;
}
```

```
Output: A3 dtor
```

```
A2 dtor
```

```
A1 dtor
```

```
B1 dtor
```

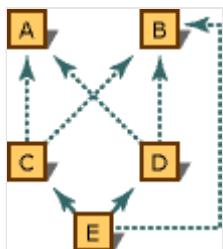
```
B3 dtor
```

```
B2 dtor
```

```
B1 dtor
```

Clases base virtuales

Los destructores de las clases base virtuales se llaman en orden inverso al de su aparición en un gráfico acíclico dirigido (recorrido con prioridad de profundidad, de izquierda a derecha y en postorden). La ilustración siguiente representa un gráfico de herencia.



A continuación se enumeran los encabezados de las clases que se muestran en la ilustración.

```
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
```

Para determinar el orden de destrucción de las clases base virtuales de un objeto de tipo `E`, el compilador compila una lista aplicando el algoritmo siguiente:

1. Recorrer el gráfico izquierdo, desde el punto más profundo del gráfico (en este caso, `E`).
2. Realizar recorridos hacia la izquierda hasta que se hayan visitado todos los nodos. Anotar el nombre del nodo actual.

3. Revisitar el nodo anterior (abajo y a la derecha) para averiguar si el nodo recordado es una clase base virtual.
4. Si el nodo recordado es una clase base virtual, examinar la lista para ver si ya se ha especificado. Si no es una clase base virtual, omitala.
5. Si el nodo recordado aún no está en la lista, agréguelo a la parte inferior de la lista.
6. Recorrer el gráfico hacia arriba y a lo largo de la ruta siguiente a la derecha.
7. Vaya al paso 2.
8. Cuando se agote la última ruta ascendente, anotar el nombre del nodo actual.
9. Vaya al paso 3.
10. Continuar este proceso hasta que el nodo inferior sea de nuevo el nodo actual.

Por consiguiente, para la clase E, el orden de destrucción es:

1. La clase base no virtual E.
2. La clase base no virtual D.
3. La clase base no virtual C.
4. La clase base virtual B.
5. La clase base virtual A.

Este proceso produce una lista ordenada de entradas únicas. Ningún nombre de clase aparece dos veces. Una vez construida la lista, se recorre en orden inverso y se llama al destructor para cada una de las clases de la lista de la última a la primera.

El orden de construcción o destrucción es principalmente importante cuando los constructores o destructores de una clase dependen del otro componente que se va a crear primero o conservar más, por ejemplo, si el destructor de A (en la figura mostrada anteriormente) se basaba en B estar presente cuando se ejecuta su código, o viceversa.

Tales interdependencias entre clases en un gráfico de herencia son inherentemente peligrosas, porque las últimas clases derivadas pueden cambiar cuál es la ruta más a la izquierda y, en consecuencia, pueden cambiar el orden de construcción y destrucción.

Clases base no virtuales

Los destructores para clases base no virtuales se invocan en orden inverso al que se declaran los nombres de clase base. Considere la siguiente declaración de clase:

C++

```
class MultInherit : public Base1, public Base2  
{  
    ...
```

En el ejemplo anterior, el destructor para `Base2` se invoca antes que el destructor para `Base1`.

Llamadas de destructor explícitas

Raras veces se necesita llamar explícitamente al destructor. Sin embargo, puede ser útil realizar la limpieza de los objetos colocados en direcciones absolutas. Estos objetos suelen asignarse mediante un operador `new` definido por el usuario que toma un argumento de ubicación. El `delete` operador no puede desasignar esta memoria porque no está asignada desde el almacén gratuito (para obtener más información, vea [Los operadores nuevos y eliminar](#)). Sin embargo, una llamada al destructor puede realizar la limpieza adecuada. Para llamar explícitamente al destructor para un objeto, `s`, de clase `String`, utilice una de las instrucciones siguientes:

C++

```
s.String::~String();      // non-virtual call  
ps->String::~String();  // non-virtual call  
  
s.~String();              // Virtual call  
ps->~String();           // Virtual call
```

La notación para las llamadas explícitas a destructores, que se muestra anteriormente, puede utilizarse con independencia de que el tipo defina un destructor. Esto permite realizar llamadas explícitas sin saber si un destructor está definido para el tipo. Una llamada explícita a un destructor donde no se ha definido ninguno no tiene ningún efecto.

Programación sólida

Una clase necesita un destructor si adquiere un recurso, y para administrar el recurso de forma segura probablemente tiene que implementar un constructor de copia y una asignación de copia.

Si el usuario no define estas funciones especiales, el compilador las define implícitamente. Los constructores y operadores de asignación generados implícitamente realizan una copia superficial miembro a miembro, que casi con total seguridad será incorrecta si un objeto está administrando un recurso.

En el ejemplo siguiente, el constructor de copia generado implícitamente hará que los punteros `str1.text` y `str2.text` hagan referencia a la misma memoria y, cuando devolvamos de `copy_strings()`, esa memoria se eliminará dos veces, lo que es un comportamiento indefinido:

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
```

La definición explícita de un destructor, un constructor de copia o un operador de asignación de copia impide la definición implícita del constructor de movimiento y el operador de asignación de movimiento. En este caso, si la copia es costosa, el hecho de no ofrecer operaciones de movimiento suele ser una oportunidad de optimización perdida.

Consulte también

[Constructores de copia y operadores de asignación de copia](#)

[Constructores de movimiento y operadores de asignación de movimiento](#)

Información general de Funciones miembro

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Las funciones miembro son estáticas o no estáticas. El comportamiento de las funciones miembro estáticas difiere del de otras funciones miembro porque las funciones miembro estáticas no tienen ningún argumento `this` implícito. Las funciones miembro no estáticas tienen un puntero `this`. Las funciones miembro, ya sean estáticas o no estáticas, pueden definirse dentro o fuera de la declaración de clase.

Si una función miembro se define dentro de una declaración de clase, se trata como una función insertada y no es necesario calificar el nombre de función con su nombre de clase. Aunque las funciones definidas dentro de declaraciones de clase ya se tratan como funciones insertadas, puede usar la palabra clave `inline` para documentar el código.

A continuación se muestra un ejemplo de declaración de una función dentro de una declaración de clase:

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()
{}
```

Si una definición de función miembro está fuera de la declaración de clase, se trata como una función insertada solo si se declara explícitamente como `inline`. Además, el nombre de función en la definición se debe calificar con su nombre de clase mediante el operador de resolución de ámbito (`::`).

El ejemplo siguiente es idéntico a la declaración anterior de clase `Account`, excepto en que la función `Deposit` se define fuera de la declaración de clase:

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()
{}
```

ⓘ Nota

Aunque las funciones miembro pueden definirse dentro de una declaración de clase o por separado, las funciones miembro se pueden agregar a una clase una vez definida la clase.

Las clases que contienen funciones miembro pueden tener muchas declaraciones, pero las funciones miembro en sí deben tener solo una definición en un programa. Varias definiciones generan un mensaje de error en tiempo de vinculación. Si una clase contiene definiciones de funciones insertadas, las definiciones de función deben ser idénticas para observar esta regla de "una definición".

virtual (Especificador)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

La palabra clave [virtual](#) solo se puede aplicar a las funciones de miembro de clase no estáticas. Eso significa que el enlace de las llamadas a la función se aplazará hasta el momento de la ejecución. Para más información, consulte [Funciones virtuales](#).

override (especificador)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Puede usar la palabra clave **override** para indicar las funciones miembro que invalidan una función virtual en una clase base.

Sintaxis

```
function-declaration override;
```

Comentarios

override es contextual y tiene un significado especial solo cuando se utiliza después de una declaración de función miembro; de lo contrario, no es una palabra clave reservada.

Ejemplo

Use **override** para evitar el comportamiento inadvertido de herencia en el código. En el ejemplo siguiente se muestra dónde puede no haberse previsto el comportamiento de la función miembro de clase derivada, sin usar **override**. El compilador no genera ningún error para este código.

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
};
```

Al usar **override**, el compilador genera errores en lugar de crear silenciosamente nuevas funciones miembro.

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()
};
```

Para especificar que las funciones no pueden reemplazarse y que las clases no se pueden heredar, use la palabra clave **final**.

Consulte también

Especificador final

Palabras clave

final (especificador)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Puede usar la palabra clave **final** para designar funciones virtuales que no se pueden invalidar en una clase derivada. También puede utilizarla para designar clases que no se pueden heredar.

Sintaxis

```
function-declaration final;  
class class-name final base-classes
```

Comentarios

final es contextual y tiene un significado especial solo cuando se usa después de una declaración de función o un nombre de clase; de lo contrario, no es una palabra clave reservada.

Cuando `final` se usa en declaraciones de clase, `base-classes` es una parte opcional de la declaración.

Ejemplo

En el ejemplo siguiente se usa la palabra clave **final** para especificar que una función virtual no se puede invalidar.

Para obtener información sobre cómo especificar que las funciones miembro se puedan invalidar, vea [override \(especificador\)](#).

En el ejemplo siguiente se usa la palabra clave **final** para especificar que una clase no se puede heredar.

C++

```
class BaseClass final
{
};

class DerivedClass: public BaseClass // compiler error: BaseClass is
                                    // marked as non-inheritable
{
```

Consulte también

[Palabras clave](#)

[override \(Especificador\)](#)

Herencia (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

En esta sección se explica cómo utilizar clases derivadas para crear programas extensibles.

Información general

Se pueden衍生新的类从现有的类使用称为“继承”的机制(参见[简单继承](#))。用于继承的类被称为“基类”或“派生类”。一个派生类通过以下语法声明：

```
C++  
  
class Derived : [virtual] [access-specifier] Base  
{  
    // member list  
};  
class Derived : [virtual] [access-specifier] Base1,  
    [virtual] [access-specifier] Base2, . . .  
{  
    // member list  
};
```

在类名(名称)之后，会出现一个冒号，后面跟着一个或多个基类的声明。声明的基类必须在类之前声明。基类声明可以包含访问说明符，即[public](#)、[protected](#)或[private](#)。访问说明符出现在类名之前。这些访问说明符仅适用于该基类。继承控制成员对基类的访问权限。有关[成员访问控制](#)的更多信息，请参阅[成员访问控制](#)。如果省略访问说明符，则将被视为[private](#)。基类声明也可以包含[virtual](#)关键字，以表示虚基类。关键字[virtual](#)可以在类名之前或之后出现。

可以指定多个基类，用逗号分隔。如果指定一个基类，则继承模型为[单继承](#)。如果指定两个或两个以上的基类，则继承模型为[多重继承](#)。

Se tratan los siguientes temas:

- Herencia única
- Varias clases base
- Funciones virtuales
- Invalidaciones explícitas
- Clases abstractas
- Resumen de reglas de ámbito

Las palabras clave `_super` e `_interface` se documentan en esta sección.

Vea también

[Referencia del lenguaje C++](#)

Funciones virtuales

Artículo • 03/03/2023 • Tiempo de lectura: 4 minutos

Una función virtual es una función miembro que se espera volver a definir en clases derivadas. Cuando se hace referencia a un objeto de clase derivada mediante un puntero o una referencia a la clase base, se puede llamar a una función virtual para ese objeto y ejecutar la versión de la clase derivada de la función.

Las funciones virtuales garantizan que se llame a la función correcta para un objeto, con independencia de la expresión utilizada para llamarla.

Suponga que una clase base contiene una función declarada como `virtual` y una clase derivada define la misma función. La función de la clase derivada se invoca para los objetos de la clase derivada, aunque se llame mediante un puntero o una referencia a la clase base. En el ejemplo siguiente se muestra una clase base que proporciona una implementación de la función `PrintBalance` y dos clases derivadas

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();
}

```

En el código anterior, las llamadas a `PrintBalance` son idénticas, excepto por el objeto al que apunta `pAccount`. Dado que `PrintBalance` es virtual, se llama a la versión de la función definida para cada objeto. La función `PrintBalance` de las clases derivadas `CheckingAccount` y `SavingsAccount` “reemplaza” la función en la clase base `Account`.

Si se declara una clase que no proporciona una implementación de reemplazo de la función `PrintBalance`, se usa la implementación predeterminada de la clase base `Account`.

Las funciones de clases derivadas reemplazan funciones virtuales de clases base solo si son del mismo tipo. Una función de una clase derivada no puede diferir de una función virtual de una clase base solo en su tipo de valor devuelto; la lista de argumentos también debe ser diferente.

Al llamar a una función mediante punteros o referencias, se aplican las siguientes reglas:

- Una llamada a una función virtual se resuelve de acuerdo con el tipo subyacente del objeto para el que se llama.
- Una llamada a una función no virtual se resuelve de acuerdo con el tipo de puntero o de referencia.

En el ejemplo siguiente se muestra cómo se comportan las funciones virtuales y no virtuales cuando se llaman mediante punteros:

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

```

Observe que, independientemente de si la función `NameOf` se invoca a través de un puntero a `Base` o un puntero a `Derived`, llama a la función para `Derived`. Llama a la función para `Derived` porque `NameOf` es una función virtual y tanto `pBase` como `pDerived` apuntan a un objeto de tipo `Derived`.

Dado que solo se llama a funciones virtuales para objetos de tipos de clase, no se puede declarar funciones globales o estáticas como `virtual`.

La palabra clave `virtual` se puede usar para declarar funciones de reemplazo en una clase derivada, pero no es necesario; los reemplazos de funciones virtuales son siempre virtuales.

Las funciones virtuales de una clase base se deben definir a menos que se declaren mediante *pure-specifier*. (Para más información sobre las funciones virtuales puras, consulte [Clases abstractas](#)).

El mecanismo de llamada a funciones virtuales se puede suprimir calificando explícitamente el nombre de función con el operador de resolución de ámbito (`::`). Considere el ejemplo anterior que implica la clase de `Account`. Para llamar a `PrintBalance` en la clase base, utilice código como el siguiente:

C++

```
CheckingAccount *pChecking = new CheckingAccount( 100.00 );  
  
pChecking->Account::PrintBalance(); // Explicit qualification.  
  
Account *pAccount = pChecking; // Call Account::PrintBalance  
  
pAccount->Account::PrintBalance(); // Explicit qualification.
```

Ambas llamadas a `PrintBalance` en el ejemplo anterior suprinen el mecanismo de llamada de función virtual.

Herencia única

Artículo • 03/03/2023 • Tiempo de lectura: 4 minutos

En la "herencia única", una forma de herencia común, las clases solo tienen una clase base. Considere la relación que se muestra en la siguiente ilustración.

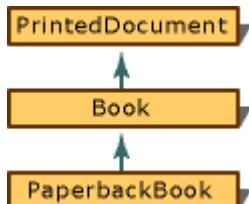


Gráfico sencillo de herencia única

Observe la progresión de general a específico en la ilustración. Otro atributo común que se encuentra en el diseño de la mayoría de jerarquías de clases es que la clase derivada tiene una "especie" de relación con la clase base. En la ilustración, `Book` es una clase de `PrintedDocument` y `PaperbackBook` es una clase de `book`.

Otro elemento de interés en la ilustración: `Book` es una clase derivada (de `PrintedDocument`) y una clase base (`PaperbackBook` se deriva de `Book`). En el ejemplo siguiente se muestra una declaración estructural de esta jerarquía de clases:

C++

```
// deriv_SingleInheritance.cpp
// compile with: /LD
class PrintedDocument {};

// Book is derived from PrintedDocument.
class Book : public PrintedDocument {};

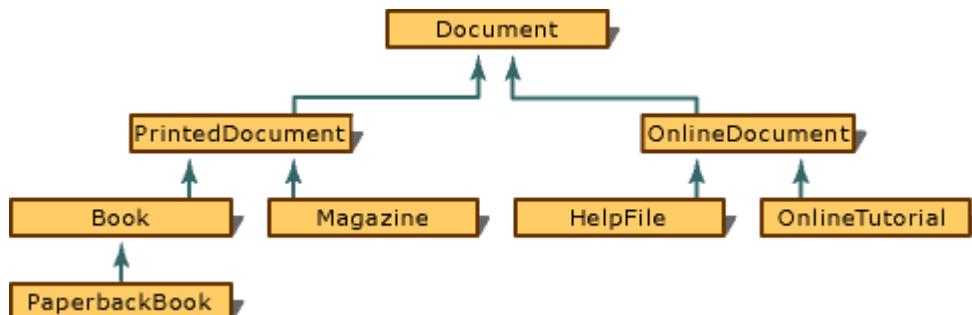
// PaperbackBook is derived from Book.
class PaperbackBook : public Book {};
```

`PrintedDocument` se considera una clase "base directa" de `Book`; es una clase "base indirecta" de `PaperbackBook`. La diferencia es que una clase base directa aparece en la lista base de una declaración de clase y una clase base indirecta no.

La clase base de la que se deriva cada clase se declara antes de la declaración de la clase derivada. No es suficiente proporcionar una declaración de referencia adelantada para una clase base; debe ser una declaración completa.

En el ejemplo anterior, se usa el especificador de acceso `public`. El significado de herencia pública, protegida y privada se describe en [Control de acceso a miembros](#).

Una clase puede actuar como clase base para muchas clases específicas, como se muestra en la ilustración siguiente.



Ejemplo de gráfico acíclico dirigido

En el diagrama anterior, denominado "gráfico acíclico dirigido" (o DAG), algunas de las clases son clases base para más de una clase derivada. Sin embargo, no sucede lo mismo al contrario: solo hay una clase base directa para una clase derivada dada. El gráfico de la ilustración muestra una estructura de "herencia única".

① Nota

Los gráficos acíclicos dirigidos no son exclusivos de la herencia única. También se usan para ilustrar gráficos de herencia múltiple.

En la herencia, la clase derivada contiene los miembros de la clase base más cualquier miembro nuevo que se agregue. Como resultado, una clase derivada puede hacer referencia a miembros de la clase base (a menos que esos miembros se redefinan en la clase derivada). Se puede usar el operador de resolución de ámbito (`::`) para hacer referencia a los miembros de clases base directas o indirectas cuando esos miembros se han vuelto a definir en la clase derivada. Considere este ejemplo:

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;
};

```

Observe que el constructor para `Book`, (`Book::Book`), tiene acceso al miembro de datos, `Name`. En un programa, se puede crear un objeto de tipo `Book`, que se usará del siguiente modo:

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();

```

Como demuestra el ejemplo anterior, los datos y funciones heredados y miembros de clase se usan de forma idéntica. Si la implementación de la clase `Book` solicita la reimplementación de la función `PrintNameOf`, la función que pertenece a la clase `Document` solo se puede llamar mediante el operador de resolución de ámbito (::):

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();
}

```

Los punteros y las referencias a clases derivadas se pueden convertir implícitamente a punteros y referencias a sus clases base si hay una clase base accesible e inequívoca. En el código siguiente se muestra este concepto mediante el uso de punteros (el mismo principio se aplica a las referencias):

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;
}

```

En el ejemplo anterior, se crean tipos diferentes. Sin embargo, dado que todos estos tipos se derivan de la clase `Document`, hay una conversión implícita a `Document *`. Como resultado, `DocLib` es una "lista heterogénea" (una lista en la que no todos los objetos son del mismo tipo) que contiene diferentes tipos de objetos.

Dado que la clase `Document` tiene una función `PrintNameOf`, puede imprimir el nombre de cada libro de la biblioteca, aunque puede omitir algo de información específica del tipo de documento (recuento de páginas de `Book`, número de bytes para `HelpFile`, etc.).

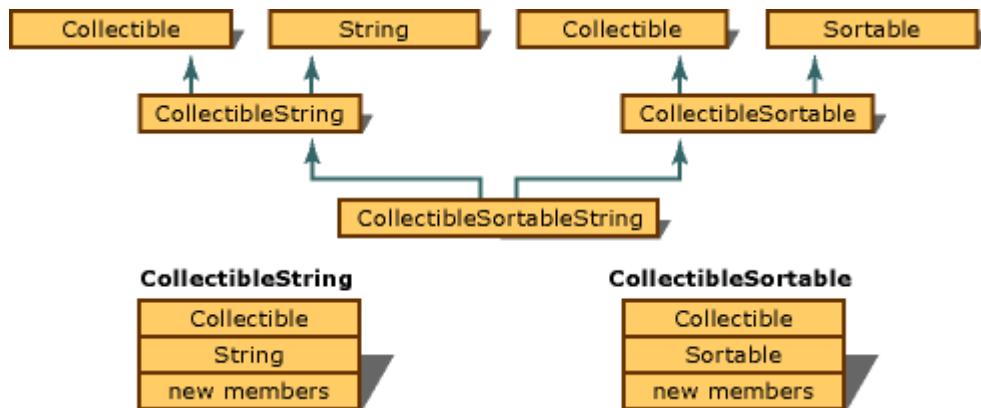
ⓘ Nota

Forzar la clase base para implementar una función como `PrintNameOf` no suele ser el mejor diseño. **Funciones virtuales** proporciona otras alternativas de diseño.

Clases base

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

El proceso de herencia crea una nueva clase derivada que se compone de los miembros de la clase o clases base más cualquier nuevo miembro agregado por la clase derivada. En una herencia múltiple, es posible crear un gráfico de herencia donde la misma clase base forme parte de varias de las clases derivadas. En la ilustración siguiente se muestra este tipo de gráfico.



Varias instancias de una clase base única

En la ilustración, se muestran las representaciones gráficas de los componentes de `CollectibleString` y `CollectibleSortable`. Sin embargo, la clase base, `Collectible`, está en `CollectibleSortableString` a través de la ruta de `CollectibleString` y la ruta de `CollectibleSortable`. Para eliminar esta redundancia, estas clases se pueden declarar como clases base virtuales cuando se heredan.

Varias clases base

Artículo • 03/03/2023 • Tiempo de lectura: 8 minutos

Una clase puede ser derivada de más de una clase base. En un modelo de herencia múltiple (en el que las clases derivan de más de una clase base), las clases base se especifican utilizando el elemento de gramática de la *lista base*. Por ejemplo, se puede especificar la declaración de clase para `CollectionOfBook`, derivada de `Collection` y `Book`:

C++

```
// deriv_MultipleBaseClasses.cpp
// compile with: /LD
class Collection {
};
class Book {};
class CollectionOfBook : public Book, public Collection {
    // New members
};
```

El orden en que se especifican las clases base no es significativo salvo en algunos casos en que se invocan constructores y destructores. En estos casos, el orden en que se especifican las clases base afecta a lo siguiente:

- El orden en que tiene lugar la inicialización mediante constructor. Si el código se basa en la parte `Book` de `CollectionOfBook` que se va inicializar antes que la parte `Collection`, el orden de especificación es significativo. La inicialización tiene lugar en el orden en que se especifican las clases en la *lista base*.
- El orden en que los destructores se invocan para la limpieza. De nuevo, si una "parte concreta" de la clase debe estar presente cuando se está destruyendo la otra parte, el orden es significativo. Los destructores se llaman en el orden inverso al de las clases especificadas en la *lista base*.

! Nota

El orden de especificación de clases base puede afectar al diseño de memoria de la clase. No se deben tomar decisiones de programación basadas en el orden de los miembros base en la memoria.

Cuando se especifica la *lista base*, no se puede especificar el mismo nombre de clase más de una vez. Sin embargo, es posible que una clase sea una base indirecta a una

clase derivada más de una vez.

Clases base virtuales

Dado que una clase puede ser una clase base indirecta de una clase derivada más de una vez, C++ proporciona una manera de optimizar el funcionamiento de esas clases base. Las clases base virtuales proporcionan una manera de ahorrar espacio y evitar la ambigüedad en las jerarquías de clases que usan la herencia múltiple.

Cada objeto no virtual contiene una copia de los miembros de datos definidos en la clase base. Esta duplicación desperdicia espacio y requiere especificar qué copia de los miembros de la clase base se desea siempre que se accede a ellos.

Cuando una clase base se especifica como base virtual, puede actuar como base indirecta más de una vez sin la duplicación de sus miembros de datos. Todas las clases base que utilizan una clase base como base virtual comparten una única copia de sus miembros de datos.

Al declarar una clase base virtual, la `virtual` palabra clave aparece en las listas base de las clases derivadas.

Considere la jerarquía de clases de la ilustración siguiente, que muestra un gráfico Lunch-Line simulado.

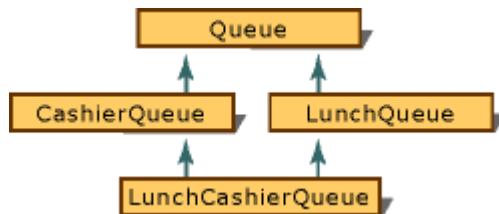
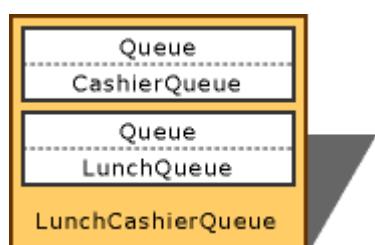


Gráfico simulado de la línea de receso

En la ilustración, `Queue` es la clase base de `CashierQueue` y `LunchQueue`. Sin embargo, cuando ambas clases se combinan para formar `LunchCashierQueue`, surge el siguiente problema: la nueva clase contiene dos subobjetos de tipo `Queue`, uno de `CashierQueue` y otro de `LunchQueue`. La ilustración siguiente muestra el diseño de memoria conceptual (el diseño de memoria real se podría optimizar).



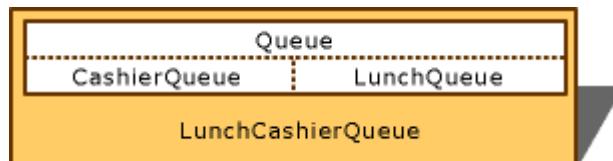
Objeto de línea de comida simulada

Observe que hay dos subobjetos `Queue` en el objeto `LunchCashierQueue`. El código siguiente declara `Queue` como clase base virtual:

C++

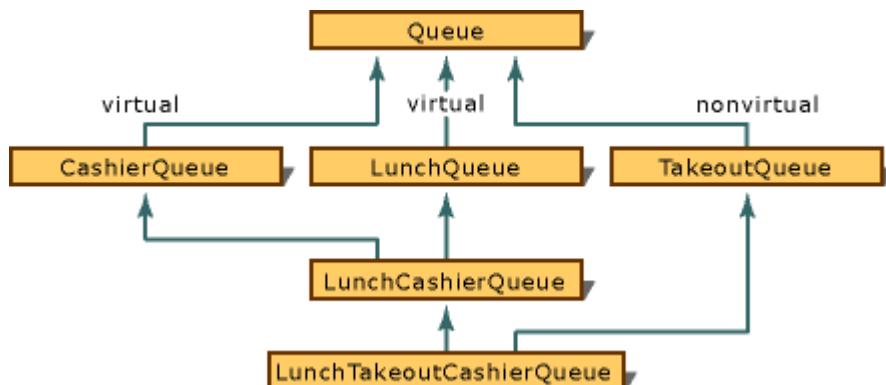
```
// deriv_VirtualBaseClasses.cpp
// compile with: /LD
class Queue {};
class CashierQueue : virtual public Queue {};
class LunchQueue : virtual public Queue {};
class LunchCashierQueue : public LunchQueue, public CashierQueue {};
```

La `virtual` palabra clave garantiza que solo se incluya una copia del subobjeto `Queue` (consulte la siguiente figura).



Objeto de línea de receso simulado con clases básicas virtuales

Una clase puede tener un componente virtual y un componente no virtual de un tipo determinado. Esto sucede en las condiciones que se muestran en la siguiente ilustración.



Componentes virtuales y no virtuales de la misma clase

En la ilustración, `CashierQueue` y `LunchQueue` usan `Queue` como clase base virtual. Sin embargo, `TakeoutQueue` especifica `Queue` como clase base, no como una clase base virtual. Por consiguiente, `LunchTakeoutCashierQueue` tiene dos subobjetos de tipo `Queue`: uno en la ruta de herencia que incluye `LunchCashierQueue` y otro en la ruta que incluye `TakeoutQueue`. Esto se muestra en la ilustración siguiente.



Diseño de objetos con herencia virtual y no virtual

ⓘ Nota

La herencia virtual supone una importante ventaja con respecto al tamaño si se compara con la herencia no virtual. Sin embargo, puede agregar una sobrecarga de procesamiento.

Si una clase derivada reemplaza una función virtual que hereda de una clase base virtual y si un constructor o destructor para la clase derivada llama a esa función con un puntero a la clase base virtual, el compilador puede incluir campos "vtordisp" ocultos adicionales en clases con bases virtuales. La `/vd0` opción del compilador suprime la adición del miembro de desplazamiento oculto del constructor/destructor vtordisp. La `/vd1` opción del compilador, por defecto, los habilita cuando son necesarios. Desactive los vtordisp solo si está seguro de que todos los constructores y destructores de clase llaman a funciones virtuales virtualmente.

La `/vd` opción del compilador afecta a todo un módulo de compilación. Utilice el `vtordisp` pragma para suprimir y volver a activar `vtordisp` los campos clase por clase:

C++

```
#pragma vtordisp( off )
class GetReal : virtual public { ... };
\#pragma vtordisp( on )
```

Ambigüedades en nombres

La herencia múltiple introduce la posibilidad de que los nombres se hereden a lo largo de más de una ruta. Los nombres de miembros de clase a lo largo de estas rutas no son necesariamente exclusivos. Estos conflictos de nombre se denominan "ambigüedades".

Cualquier expresión que haga referencia a un miembro de clase debe producir una referencia ambigua. En el ejemplo siguiente se muestra cómo se desarrollan las ambigüedades:

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(); // Note that 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 {};

```

Dadas las declaraciones de clase anteriores, un código como el siguiente es ambiguo porque no está claro si `b` hace referencia a `b` en `A` o en `B`:

C++

```

C *pc = new C;

pc->b();

```

Considere el ejemplo anterior. Dado que el nombre `a` es miembro de la clase `A` y la clase `B`, el compilador no puede discernir qué `a` designa la función que se va a invocar. El acceso a un miembro es ambiguo si puede hacer referencia a más de una función, objeto, tipo o enumerador.

El compilador detecta las ambigüedades realizando pruebas en este orden:

1. Si el acceso al nombre es ambiguo (como se acaba de describir), se genera un mensaje de error.
2. Si las funciones sobrecargadas no son ambiguas, se resuelven.
3. Si el acceso al nombre infringe el permiso de acceso a miembros, se genera un mensaje de error. (para obtener más información, consulte [Control de acceso de los miembros](#)).

Cuando una expresión produce una ambigüedad en la herencia, la puede resolver manualmente calificando el nombre en cuestión con su nombre de clase. Para que la

compilación del ejemplo anterior se realice correctamente sin ambigüedades, utilice código como el siguiente:

C++

```
C *pc = new C;  
pc->B::a();
```

ⓘ Nota

Cuando se declara `C`, tiene la posibilidad de producir errores cuando se hace referencia a `B` en el ámbito de `C`. Sin embargo, no se emite ningún error hasta se realice realmente una referencia no calificada a `B` en el ámbito de `C`.

Dominación

Es posible que se llegue a varios nombres (función, objeto o enumerador) a través de un gráfico de herencia. Estos casos se consideran ambiguos con las clases base no virtuales. También son ambiguos con las clases base virtuales, a menos que uno de los nombres "domine" a los otros.

Un nombre domina a otro nombre si está definido en ambas clases y una clase se deriva de la otra. El nombre dominante es el nombre de la clase derivada; este nombre se utiliza cuando podría producirse una ambigüedad, como se muestra en el ejemplo siguiente:

C++

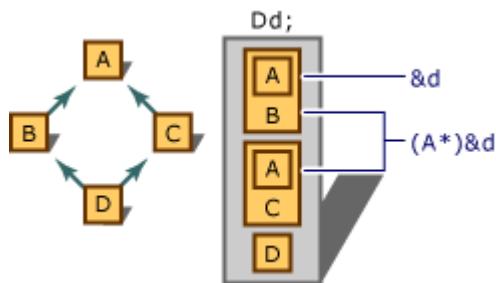
```
// deriv_Dominance.cpp  
// compile with: /LD  
class A {  
public:  
    int a;  
};  
  
class B : public virtual A {  
public:  
    int a();  
};  
  
class C : public virtual A {};  
  
class D : public B, public C {  
public:
```

```
D() { a(); } // Not ambiguous. B::a() dominates A::a.  
};
```

Conversiones ambiguas

Las conversiones explícitas e implícitas de punteros o referencias a los tipos de clase pueden producir ambigüedades. En la ilustración siguiente, Conversión ambigua de punteros a clases base, se muestra lo siguiente:

- La declaración de un objeto de tipo `D`.
- El efecto de aplicar el operador de la dirección (`&`) a ese objeto. Observe que el operador address-of siempre proporciona la dirección base del objeto.
- El efecto de convertir explícitamente el puntero obtenido mediante el operador address-of al tipo de clase base `A`. Observe que forzar la dirección del objeto al tipo `A*` no siempre proporciona al compilador la información suficiente para determinar qué objeto secundario de tipo `A` debe seleccionar; en este caso, existen dos objetos secundarios.



Conversión ambigua de punteros a clases base

La conversión al tipo `A*` (puntero a `A`) es ambigua porque no hay ninguna manera de discernir qué objeto secundario de tipo `A` es el correcto. Observe que puede evitar la ambigüedad si especifica explícitamente el objeto secundario que quiere utilizar, como sigue:

C++

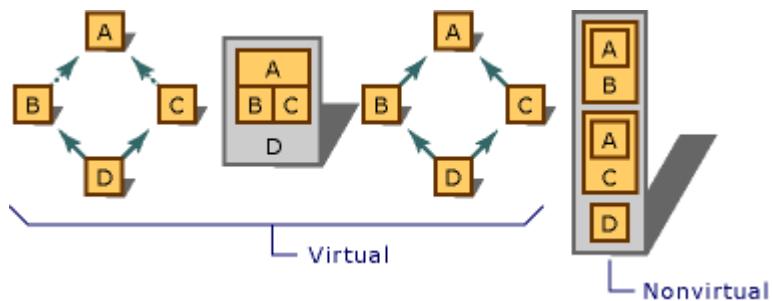
```
(A *)(B *)&d      // Use B subobject.  
(A *)(C *)&d      // Use C subobject.
```

Ambigüedades y clases base virtuales

Si se utilizan clases base virtuales, se puede tener acceso a las funciones, objetos, tipos y enumeradores a través de rutas de herencia múltiple. Como solo hay una instancia de la

clase base, no se produce ambigüedad a la hora de acceder a estos nombres.

En la ilustración siguiente se muestra cómo se componen los objetos mediante herencia virtual y no virtual.



Derivación virtual y no virtual

En la ilustración, el acceso a cualquier miembro de la clase **A** a través de clases base no virtuales produce ambigüedad; el compilador no tiene información que explique si se debe usar el subobjeto asociado a **B** o el subobjeto asociado a **C**. Sin embargo, cuando se especifica **A** como una clase base virtual, no hay dudas acerca de a qué subobjeto se está accediendo.

Consulte también

[Herencia](#)

Invalidaciones explícitas (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Si la misma función virtual se declara en dos o más [interfaces](#) y una clase se deriva de estas interfaces, puede invalidar explícitamente cada función virtual.

Para más información sobre las invalidaciones explícitas en código administrado mediante C++/CLI, consulte [Invalidaciones explícitas](#).

FIN de Específicos de Microsoft

Ejemplo

En el ejemplo de código siguiente se muestra cómo utilizar las invalidaciones explícitas:

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)
```

Consulte también

[Herencia](#)

Clases abstractas (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Las clases abstractas actúan como expresiones de conceptos generales de los que pueden derivarse clases más concretas. No se puede crear un objeto de un tipo de clase abstracta, pero se pueden usar punteros y referencias a tipos de clase abstracta.

Para crear una clase abstracta, declare al menos una función miembro virtual pura. Se trata de una función virtual declarada mediante la sintaxis del especificador *puro* ($= 0$). Las clases derivadas de la clase abstracta deben implementar la función virtual pura o deben ser también clases abstractas.

Considere el ejemplo que se presenta en [Funciones virtuales](#). El propósito de la clase `Account` es proporcionar funcionalidad general, pero los objetos de tipo `Account` son demasiado generales para resultar útiles. Eso significa que `Account` es un buen candidato para una clase abstracta:

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;
};
```

La única diferencia entre esta declaración y la anterior consiste en que `PrintBalance` se declara con el especificador puro ($= 0$).

Restricciones para el uso de clases abstractas

No se pueden usar clases abstractas para:

- Variables o datos de miembro
- Tipos de argumento
- Tipos de valor devuelto de función
- Tipos de conversiones explícitas

Si el constructor para una clase abstracta llama a una función virtual pura, ya sea directa o indirectamente, el resultado es indefinido. Sin embargo, los constructores y destructores para las clases abstractas pueden llamar a otras funciones miembro.

Funciones virtuales puras definidas

En las clases abstractas, las funciones virtuales puras pueden estar *definidas* o tener una implementación. Solo se puede llamar a estas funciones mediante la sintaxis completa:

abstract-class-name::function-name()

Las funciones virtuales puras definidas son útiles al diseñar jerarquías de clases cuyas clases base incluyen destructores virtuales puros. Esto se debe a que siempre se llama a los destructores de clase base durante la destrucción de objetos. Considere el ejemplo siguiente:

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
}
```

En el ejemplo se muestra la forma en que una extensión del compilador de Microsoft permite agregar una definición insertada a un objeto `~base()` virtual puro. También se puede definir fuera de la clase mediante `base::~base() {}`.

Cuando el objeto `aDerived` sale del ámbito, se llama al destructor de la clase `derived`. El compilador genera código para llamar implícitamente al destructor para la clase `base` después del destructor `derived`. La implementación vacía para el objeto `~base` de la función virtual pura garantiza que al menos existe alguna implementación para la función. Sin él, el enlazador genera un error de símbolo externo sin resolver para la llamada implícita.

ⓘ Nota

En el ejemplo anterior, la función virtual pura `base::~base` se llama implícitamente desde `derived::~derived`. También es posible llamar a funciones virtuales puras explícitamente mediante el nombre completo de la función miembro. Estas funciones deben tener una implementación; de lo contrario, la llamada produce un error en tiempo de vínculo.

Consulte también

[Herencia](#)

Resumen de reglas de ámbito

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

El uso de un nombre debe ser inequívoco dentro de su ámbito (hasta el punto en que se determina la sobrecarga). Si el nombre indica una función, la función no debe ser ambigua respecto al número y tipo de parámetros. Si el nombre se mantiene no ambiguo, se aplican las reglas de [acceso a miembros](#).

Inicializadores del constructor

Los [inicializadores de constructor](#) se evalúan en el ámbito del bloque más externo del constructor para el que se especifican. Por lo tanto, pueden usar los nombres de parámetro del constructor.

Nombres globales

Un nombre de un objeto, una función o un enumerador es global si se presenta fuera de cualquier función o clase o está precedido por el operador unario global de ámbito (:) y si no se usa junto con alguno de estos operadores binarios:

- Resolución de ámbito (:)
- Selección de miembro para objetos y referencias (.)
- Selección de miembro para punteros (->)

Nombres completos

Los nombres utilizados con el operador binario de resolución de ámbito (:) se denominan “nombres completos”. El nombre especificado detrás del operador binario de resolución de ámbito debe ser un miembro de la clase especificada a la izquierda del operador o un miembro de su clase o clases base.

Los nombres especificados detrás del operador de selección de miembro (. o ->) deben ser miembros del tipo de clase del objeto especificado a la izquierda del operador o miembros de su clase o clases base. Los nombres especificados a la derecha del operador de selección de miembro (->) también pueden ser objetos de otro tipo de clase, siempre que el lado izquierdo de -> sea un objeto de clase y que la clase defina un operador de selección de miembro sobrecargado (->) que se evalúe como un

puntero a otro tipo de clase. (Esta especificación se explica con más detalle en [Acceso a miembros de clase](#)).

El compilador busca los nombres en el orden siguiente y se detiene cuando encuentra el nombre:

1. El ámbito de bloque actual si el nombre se utiliza dentro de una función; en caso contrario, el ámbito global.
2. En el exterior, a través de cada ámbito de bloque contenedor, incluido el ámbito de función más externo (que incluye parámetros de función).
3. Si el nombre se utiliza dentro de una función miembro, se busca el nombre en el ámbito de la clase.
4. El nombre se busca en las clases base de la clase.
5. Se busca en el ámbito de la clase anidada contenedora y en sus clases base. La búsqueda continúa hasta que se busque en el ámbito de la clase contenedora más externo.
6. Se busca en el ámbito global.

Sin embargo, puede modificar este orden de búsqueda de la forma siguiente:

1. Los nombres precedidos por `::` obligan a que la búsqueda se inicie en el ámbito global.
2. Los nombres precedidos por las palabras clave `class`, `struct` y `union` fuerzan al compilador a buscar solo los nombres de `class`, `struct` o `union`.
3. Los nombres del lado izquierdo del operador de resolución de ámbito (`::`) solo pueden ser nombres `class`, `struct`, `namespace` o `union`.

Si el nombre hace referencia a un miembro no estático pero se utiliza en una función miembro estática, se genera un mensaje de error. De igual forma, si el nombre hace referencia a algún miembro no estático en una clase contenedora, se genera un mensaje de error, porque las clases contenidas no tienen punteros `this` de la clase contenedora.

Nombres de parámetro de la función

Los nombres de los parámetros de función en las definiciones de función se consideran que están en el ámbito del bloque más externo de la función. Por consiguiente, son nombres locales y salen del ámbito cuando termina la función.

Los nombres de los parámetros de función en las declaraciones de función (prototipos) están en el ámbito local de la declaración y salen del ámbito al final de la declaración.

Los parámetros predeterminados están en el ámbito del parámetro para el que son el parámetro predeterminado, como se describe en los dos párrafos anteriores. Sin embargo, no pueden acceder a variables locales o miembros de clase no estáticos. Los parámetros predeterminados se evalúan en el punto de la llamada de función, pero se evalúan en el ámbito original de la declaración de función. Por tanto, los parámetros predeterminados de funciones miembro siempre se evalúan en el ámbito de la clase.

Consulte también

[Herencia](#)

Palabras clave de herencia

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

```
class class-name  
class __single_inheritance class-name  
class __multiple_inheritance class-name  
class __virtual_inheritance class-name
```

donde:

class-name

Nombre de la clase que se está declarando.

C++ permite declarar un puntero a un miembro de clase antes de la definición de clase. Por ejemplo:

C++

```
class S;  
int S::*p;
```

En el código anterior, `p` se declara como un puntero al miembro entero de la clase `S`, pero `class S` aún no se ha definido en este código; solo se ha declarado. Cuando el compilador encuentra un puntero así, debe crear una representación generalizada del puntero. El tamaño de la representación depende del modelo de herencia especificado. Hay tres maneras de especificar un modelo de herencia al compilador:

- En la línea de comandos mediante el modificador [/vmg](#).
- Mediante el uso de la pragma [pointers_to_members](#).
- Mediante las palabras clave de herencia `__single_inheritance`, `__multiple_inheritance` y `__virtual_inheritance`. Esta técnica controla el modelo de herencia clase por clase.

! Nota

Si siempre se declara un puntero a un miembro de una clase después de definir la clase, no se necesita usar ninguna de estas opciones.

Si declara un puntero a un miembro de clase antes de definir la clase, esto puede afectar negativamente al tamaño y la velocidad del archivo ejecutable resultante. Cuanto más compleja es la herencia usada por una clase, mayor será el número de bytes necesarios para representar un puntero a un miembro de la clase y mayor será el código necesario para interpretar el puntero. La herencia única (o su ausencia) es la menos compleja, mientras que la herencia virtual es la más compleja. Los punteros a los miembros que declare antes de definir la clase siempre usan la representación más grande y compleja.

Si se cambia el ejemplo anterior a:

C++

```
class __single_inheritance S;  
int S::*p;
```

independientemente de las opciones de la línea de comandos o las pragmas que se especifiquen, los punteros a miembros de `class S` usarán la representación más pequeña posible.

ⓘ Nota

La misma declaración adelantada de una representación de puntero a miembro de la clase debe aparecer en cada unidad de traducción que declare punteros a miembros de esa clase y la declaración debe aparecer antes de que se declaren los punteros a miembros.

A efectos de compatibilidad con versiones anteriores, `_single_inheritance`, `_multiple_inheritance` y `_virtual_inheritance` son sinónimos de `_single_inheritance`, `_multiple_inheritance` y `_virtual_inheritance`, a menos que se especifique la opción del compilador/[Za \(deshabilitar extensiones de lenguaje\)](#).

FIN de Específicos de Microsoft

Consulte también

[Palabras clave](#)

virtual (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

La palabra clave **virtual** declara una función virtual o una clase base virtual.

Sintaxis

```
virtual [type-specifiers] member-function-declarator  
virtual [access-specifier] base-class-name
```

Parámetros

type-specifiers

Especifica el tipo de valor devuelto de la función miembro virtual.

member-function-declarator

Declara una función miembro.

access-specifier

Define el nivel de acceso a la clase base: **public**, **protected** o **private**. Puede aparecer antes o después de la palabra clave **virtual**.

base-class-name

Identifica un tipo de clase declarado previamente.

Comentarios

Vea [Funciones virtuales](#) para más información.

Vea también las siguientes palabras clave: [class](#), [private](#), [public](#) y [protected](#).

Consulte también

[Palabras clave](#)

__super

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Permite establecer explícitamente que se está llamando a una implementación de la clase base para una función que se va a reemplazar.

Sintaxis

```
__super::member_function();
```

Comentarios

Todos los métodos accesibles de la clase base se consideran durante la fase de la resolución de sobrecarga y la función que proporciona la mejor coincidencia es la que se llama.

__super solo puede aparecer en el cuerpo de una función miembro.

__super no se puede utilizar con una declaración using. Consulte [Usar declaración](#) para obtener más información.

Con la introducción de [atributos](#) que insertan código, el código podría contener una o más clases base cuyos nombres puede no conocer, pero que contienen métodos a los que desea llamar.

Ejemplo

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)
    }
};
```

FIN de Específicos de Microsoft

Consulte también

[Palabras clave](#)

__interface

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Una interfaz de Microsoft C++ se puede definir de la manera siguiente:

- Puede heredar de cero o más interfaces base.
- No puede heredar de una clase base.
- Solo puede contener métodos virtuales puros, públicos.
- No puede contener constructores, destructores ni operadores.
- No puede contener métodos estáticos.
- No puede contener miembros de datos; se permiten las propiedades.

Sintaxis

```
modifier __interface interface-name {interface-definition};
```

Comentarios

Una [clase](#) o [estructura](#) de C++ se podría implementar con estas reglas, pero [__interface](#) las exige.

Por ejemplo, este es un ejemplo de definición de interfaz:

C++

```
__interface IMyInterface {
    HRESULT CommitX();
    HRESULT get_X(BSTR* pbstrName);
};
```

Para obtener información sobre interfaces administradas, consulte la [clase de interfaz](#).

Observe que no tiene que indicar explícitamente que las funciones `CommitX` y `get_X` son virtuales puras. Una declaración equivalente para la primera función sería:

C++

```
virtual HRESULT CommitX() = 0;
```

`__interface` implica el modificador `novtable __declspec`.

Ejemplo

En el ejemplo siguiente se muestra cómo utilizar propiedades declaradas en una interfaz.

C++

```
// deriv_interface.cpp
#define _ATL_ATTRIBUTES 1
#include <atlbase.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

```

FIN de Específicos de Microsoft

Consulte también

[Palabras clave](#)

[Atributos de interfaz](#)

Funciones miembro especiales

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Las *funciones miembro especiales* son funciones miembro de clase (o estructura) que, en determinados casos, el compilador genera automáticamente. Estas funciones son el [constructor predeterminado](#), el [destructor](#), el [constructor de copia](#) y el [operador de asignación de copia](#), y el [constructor de movimiento](#) y el [operador de asignación de movimiento](#). Si la clase no define una o varias funciones miembro especiales, el compilador puede declarar y definir implícitamente las funciones que se usan. Las implementaciones generadas por el compilador se denominan funciones miembro especiales *predeterminadas*. El compilador no genera funciones si no son necesarias.

Puede declarar explícitamente una función miembro especial predeterminada mediante la palabra clave = **default**. Esto hace que el compilador defina la función solo si es necesario, igual que si la función no se hubiera declarado en absoluto.

En algunos casos, el compilador puede generar funciones miembro especiales *eliminadas*, que no están definidas y, por lo tanto, no se pueden llamar. Esto puede ocurrir en casos en los que una llamada a una función miembro especial determinada en una clase no tiene sentido, debido a las demás propiedades de la clase. Para evitar explícitamente la generación automática de una función miembro especial, puede declararla como eliminada mediante la palabra clave = **delete**.

El compilador genera un *constructor predeterminado*, que es un constructor que no toma ningún argumento, solo cuando no se ha declarado ningún otro constructor. Si ha declarado solo un constructor que toma parámetros, el código que intenta llamar a un constructor predeterminado hace que el compilador genere un mensaje de error. El constructor predeterminado generado por el compilador realiza una [inicialización predeterminada](#) simple miembro a miembro del objeto. La inicialización predeterminada deja todas las variables miembro en un estado indeterminado.

El destructor predeterminado realiza la destrucción miembro a miembro del objeto. Solo es virtual si un destructor de clase base es virtual.

Las operaciones predeterminadas de construcción y asignación de copia y movimiento realizan copias de patrones de bits miembro a miembro o movimientos de miembros de datos no estáticos. Las operaciones de movimiento solo se generan cuando no se declara ningún destructor ni operaciones de movimiento o copia. Un constructor de copia predeterminado solo se genera cuando no se declara ningún constructor de copia. Se elimina implícitamente si se declara una operación de movimiento. Un operador de asignación de copia predeterminado solo se genera cuando no se declara

explícitamente ningún operador de asignación de copia. Se elimina implícitamente si se declara una operación de movimiento.

Vea también

[Referencia del lenguaje C++](#)

Miembros estáticos (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Las clases pueden contener datos de miembro y funciones miembro estáticos. Cuando un miembro de datos se declara como `static`, solo se conserva una copia de los datos para todos los objetos de la clase.

Los miembros de datos estáticos no forman parte de los objetos de un tipo de clase determinado. Por tanto, la declaración de un miembro de datos estático no se considera una definición. El miembro de datos se declara en el ámbito de la clase, pero la definición se realiza en el ámbito de archivo. Estos miembros estáticos tienen vinculación externa. Esto se ilustra en el ejemplo siguiente:

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()
{}
```

En el código anterior, el miembro `bytecount` se declara en la clase `BufferedOutput`, pero debe definirse fuera de la declaración de clase.

Se puede hacer referencia a miembros de datos estáticos sin hacer referencia a un objeto de tipo de clase. El número de bytes escritos mediante objetos de

`BufferedOutput` puede obtenerse de la manera siguiente:

C++

```
long nBytes = BufferedOutput::bytecount;
```

Para que el miembro estático exista, no es necesario que exista ningún objeto del tipo de clase. También se puede obtener acceso a los miembros estáticos mediante los operadores de selección de miembros (. y ->). Por ejemplo:

C++

```
BufferedOutput Console;  
long nBytes = Console.bytecount;
```

En el caso anterior, la referencia al objeto (`Console`) no se evalúa; el valor devuelto es el del objeto estático `bytecount`.

Los miembros de datos estáticos están sujetos a las reglas de acceso a miembros de clase, por lo que el acceso privado a miembros de datos estáticos solo se permite para las funciones miembro de clase y los elementos friend. Estas reglas se describen en [Control de acceso a miembros](#). La excepción es que los miembros de datos estáticos se deben definir en el ámbito de archivo, independientemente de sus restricciones de acceso. Si el miembro de datos se va a inicializar explícitamente, se debe proporcionar un inicializador con la definición.

El nombre de clase no califica el tipo de un miembro estático. Por tanto, el tipo de `BufferedOutput::bytecount` es `long`.

Consulte también

[Clases y structs](#)

Conversiones de tipos definidos por el usuario (C++)

Artículo • 26/09/2022 • Tiempo de lectura: 11 minutos

Una *conversión* produce un valor nuevo de cierto tipo a partir de un valor de otro tipo. El lenguaje C++ tiene *conversiones estándar* integradas que son compatibles con sus tipos integrados. Se pueden crear *conversiones definidas por el usuario* para realizar conversiones a o desde tipos definidos por el usuario, o entre ellos.

Las conversiones estándar realizan conversiones entre tipos integrados, entre punteros o referencias a tipos relacionados por herencia, a y desde punteros void, y al puntero nulo. Para obtener más información, consulte [Conversiones estándar](#). Las conversiones definidas por el usuario realizan conversiones entre tipos definidos por el usuario, o entre tipos definidos por el usuario y tipos integrados. Puede implementarlas como [constructores de conversión](#) o como [funciones de conversión](#).

Las conversiones pueden ser explícitas, si el programador solicita que un tipo se convierta en otro, como en el caso de una conversión o inicialización directa, o implícitas, si el lenguaje o programa llama a un tipo que no es el determinado por el programador.

Se intenta realizar conversiones implícitas cuando:

- El argumento que se proporciona a una función no tiene el mismo tipo que el parámetro correspondiente.
- El valor que devuelve una función no tiene el mismo tipo que el tipo devuelto de la función.
- Una expresión de inicializador no tiene el mismo tipo que el objeto que está inicializando.
- Una expresión que controla una instrucción condicional, una construcción en bucle o un modificador no tiene el tipo de resultado necesario para controlarlos.
- El operando proporcionado a un operador no tiene el mismo tipo que el parámetro-operando correspondiente. En el caso de los operadores integrados, los dos operandos deben tener el mismo tipo y los dos se convierten a un tipo común que pueda representarlos a ambos. Para obtener más información, consulte [Conversiones estándar](#). En el caso de los operadores definidos por el usuario, cada operando debe tener el mismo tipo que el parámetro-operando correspondiente.

Si una conversión estándar no puede llevar a cabo una conversión implícita, el compilador puede usar una conversión definida por el usuario, que puede ir seguida de una conversión estándar adicional, para terminarla.

Si hay disponibles dos o más conversiones definidas por el usuario que realizan la misma conversión en un lugar de conversión, se dice que la conversión es ambigua. Ese tipo de ambigüedades es un error, ya que el compilador no puede determinar cuál de las conversiones disponibles debe elegir. Con todo, no constituye un error simplemente definir varias formas de realizar la misma conversión, porque el conjunto de conversiones disponibles puede ser distinto en distintos lugares del código fuente, por ejemplo, en función de los archivos de encabezados incluidos en un archivo de código fuente. Siempre que solo haya una conversión disponible en el lugar de la conversión, no hay ambigüedad. Las conversiones ambiguas pueden deberse a varios motivos, pero los más habituales son:

- Herencia múltiple. La conversión está definida en más de una clase base.
- Llamada de función ambigua. La conversión se define como constructor de conversión del tipo de destino y como función de conversión del tipo de origen.
Para obtener más información, consulte [Funciones de conversión](#).

Normalmente, las ambigüedades se pueden resolver simplemente calificando el tipo en cuestión de forma más precisa o realizando una conversión explícita que aclare su finalidad.

Los constructores y las funciones de conversión siguen reglas de control de acceso de los miembros, pero la accesibilidad de las conversiones solo se tiene en cuenta si se puede determinar una conversión no ambigua. Dicho de otro modo, una conversión puede ser ambigua incluso cuando el nivel de acceso de una conversión competidora pudiera impedir que se usara. Para obtener más información sobre la accesibilidad de los miembros, consulte [Control de acceso a miembros](#).

Palabra clave explícita y problemas con conversiones implícitas

De forma predeterminada, cuando se crea una conversión definida por el usuario, el compilador puede usarla para realizar conversiones implícitas. Hay casos en los que se desea hacer así, pero en otros las sencillas reglas que indican al compilador cómo hacer las conversiones implícitas pueden hacerle aceptar código que no se desea usar.

Un ejemplo bien conocido de conversión implícita que puede dar problemas es la conversión a `bool`. Existen muchas razones por las que podría querer crear un tipo de

clase que se pueda usar en un contexto booleano (por ejemplo, para que se pueda usar para controlar una instrucción o bucle `if`), pero cuando el compilador realiza una conversión definida por el usuario en un tipo integrado, se permite que el compilador aplique una conversión estándar adicional a continuación. El objetivo de esa conversión estándar adicional es admitir, entre otras cosas, la promoción de `short` a `int`. Además, puede admitir otras conversiones menos obvias (por ejemplo, de `bool` a `int`), lo que permite usar el tipo de clase en contextos de enteros que no se habían previsto. Este problema concreto se conoce como *problema de valor booleano seguro*. Y este es el tipo de problema con el que la palabra clave `explicit` puede ayudar.

La palabra clave `explicit` indica al compilador que la conversión especificada no se puede usar para realizar conversiones implícitas. Antes de que se introdujera la palabra clave `explicit`, si quería usar la comodidad sintáctica de las conversiones implícitas, tenía que aceptar las consecuencias no previstas que a veces creaba la conversión implícita, o bien usar como solución las funciones de conversión con nombre, que resultan menos prácticas. Ahora, al usar la palabra clave `explicit`, puede crear conversiones prácticas que solo se pueden usar para realizar conversiones explícitas o inicialización directa, lo que no da lugar al tipo de problemas que ilustra el problema de valor booleano seguro.

La palabra clave `explicit` se puede aplicar a constructores de conversión desde C++98 y a funciones de conversión desde C++11. En las secciones siguientes se proporciona más información sobre el uso de la palabra clave `explicit`.

Constructores de conversión

Los constructores de conversión definen conversiones de tipos integrados o definidos por el usuario en un tipo definido por el usuario. En el ejemplo siguiente se muestra un constructor de conversión que convierte el tipo integrado `double` en un tipo `Money` definido por el usuario.

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;
}

```

Observe que la primera llamada a la función `display_balance`, que toma un argumento de tipo `Money`, no requiere conversión porque su argumento es del tipo correcto. Aun así, en la segunda llamada a `display_balance` se necesita una conversión porque el tipo del argumento, un tipo `double` con el valor de `49.95`, no es lo que espera la función. La función no puede usar este valor directamente, pero como hay una conversión del tipo de argumento (`double`) al tipo del parámetro correspondiente (`Money`), se crea a partir del argumento un valor temporal de tipo `Money`, que se usa para finalizar la llamada a la función. Observe que, en la tercera llamada a `display_balance`, el argumento no es de tipo `double`, sino que es de tipo `float` con el valor de `9.99`. A pesar de ello, la llamada a la función se puede finalizar porque el compilador puede realizar una conversión estándar (en este caso de `float` a `double`) y, luego, realizar la conversión definida por el usuario de `double` a `Money` para terminar la conversión necesaria.

Declarar constructores de conversión

Al declarar un constructor de conversión se aplican las reglas siguientes:

- El tipo de destino de la conversión es el tipo definido por el usuario que se está construyendo.
- Generalmente, los constructores de conversión toman exactamente un argumento, que tiene el tipo del origen. Con todo, un constructor de conversión puede especificar parámetros adicionales si cada uno de ellos tiene un valor predeterminado. El tipo del primer parámetro es el tipo de origen.
- Los constructores de conversión, como todos los constructores, no especifican un tipo de retorno. La especificación de un tipo de retorno en la declaración es un error.

- Los constructores de conversión pueden ser explícitos.

Constructores de conversión explícitos

Si se declara que un constructor de conversión es `explicit`, solo se puede usar para realizar la inicialización directa de un objeto o para realizar una conversión explícita. Así se impide que las funciones que aceptan un argumento del tipo de la clase acepten también implícitamente argumentos del tipo del origen del constructor de conversión. También se impide que se inicialice una copia del tipo de la clase a partir de un valor del tipo de origen. En el ejemplo siguiente se muestra cómo definir un constructor de construcción explícito y su efecto en qué código está formado correctamente.

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;
}
```

Observe que, en este ejemplo, se puede seguir usando el constructor de conversión para realizar la inicialización directa de `payable`. Si, en lugar de ello, se inicializara una copia de `Money payable = 79.99;`, sería un error. La primera llamada a `display_balance` no se ve afectada porque el argumento es del tipo correcto. La segunda llamada a `display_balance` es un error, porque el constructor de conversión no se puede usar para

realizar conversiones implícitas. La tercera llamada a `display_balance` es válida por la presencia de la conversión explícita a `Money`, pero con todo el compilador contribuyó a la finalización de la conversión mediante la inserción de una conversión implícita de `float` a `double`.

El hecho de admitir conversiones implícitas puede parecer práctico, pero podría introducir errores difíciles de detectar. En general, es mejor que todos los constructores de conversión sean explícitos, excepto cuando se desea que una conversión concreta tenga lugar de forma implícita.

Funciones de conversión

Las funciones de conversión definen conversiones de un tipo definido por el usuario a otros tipos. Estas funciones se denominan a veces "operadores de conversión" ya que, junto con los constructores de conversión, se les llama cuando un valor se convierte en otro tipo. En el ejemplo siguiente se muestra una función de conversión que convierte el tipo definido por el usuario `Money` en un tipo `double` integrado:

```
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;  
}
```

Observe que la variable miembro `amount` se hace privada y que se introduce una función de conversión pública en el tipo `double` simplemente para que devuelva el valor de `amount`. En la función `display_balance` se produce una conversión implícita cuando el valor de `balance` se envía a una salida estándar mediante el uso del operador de inserción de secuencia `<<`. Como no se define un operador de inserción de secuencia para el tipo definido por el usuario `Money`, pero sí hay uno para el tipo integrado

`double`, el compilador puede usar la función de conversión de `Money` en `double` para cumplir con el operador de inserción de secuencia.

Las clases derivadas heredan las funciones de conversión. Las funciones de conversión de una clase derivada solo invalidan una función de conversión heredada cuando se convierten exactamente en el mismo tipo. Por ejemplo, una función de conversión definida por el usuario de la clase derivada `operator int` no invalida una función de conversión definida por el usuario de la clase base `operator short` ni le afecta de ninguna forma, a pesar de que las conversiones estándar definen una relación de conversión entre `int` y `short`.

Declarar funciones de conversión

Al declarar una función de conversión se aplican las reglas siguientes:

- El tipo de destino de la conversión de debe declarar antes que la declaración de la función de conversión. No se pueden declarar clases, estructuras, enumeraciones ni definiciones de tipos en la declaración de la función de conversión.

C++

```
operator struct String { char string_storage; }() // illegal
```

- Las funciones de conversión no toman ningún argumento. La especificación de cualquier parámetro en la declaración es un error.
- Las funciones de conversión tienen un tipo de retorno que se especifica mediante el nombre de la función de conversión, que es también el nombre del tipo de destino de la conversión. La especificación de un tipo de retorno en la declaración es un error.
- Las funciones de conversión pueden ser virtuales.
- Las funciones de conversión pueden ser explícitas.

Funciones de conversión explícita

Si se declara que una función de conversión es explícita, solo se puede usar para realizar una conversión explícita. Así se impide que las funciones que aceptan un argumento del tipo de destino de la función de conversión acepten también implícitamente argumentos del tipo de clase. También se impide que se inicialicen copias de instancias del tipo de destino a partir de un valor del tipo de clase. En el ejemplo siguiente se

muestra cómo definir una función de construcción explícita y su efecto en qué código está formado correctamente.

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;
}
```

Aquí, la función de conversión `operator double` se ha hecho explícita, y se ha introducido una conversión explícita al tipo `double` en la función `display_balance` para realizar la conversión. Si se omitiera esta conversión, el compilador no encontraría un operador de inserción de secuencia `<<` adecuado para el tipo `Money` y se produciría un error.

Miembros de datos mutables (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Esta palabra clave solo se puede aplicar a los miembros de datos no estáticos y no constantes de una clase. Si se declara un miembro de datos `mutable`, después se permite asignar un valor a este miembro de datos desde una función miembro `const`.

Sintaxis

```
mutable member-variable-declaration;
```

Comentarios

Por ejemplo, el siguiente código se compilará sin errores porque `m_accessCount` se ha declarado como `mutable` y, por tanto, se puede modificar con `GetFlag` aunque `GetFlag` sea una función miembro de tipo const.

C++

```
// mutable.cpp
class X
{
public:
    bool GetFlag() const
    {
        m_accessCount++;
        return m_flag;
    }
private:
    bool m_flag;
    mutable int m_accessCount;
};

int main()
{}
```

Consulte también

[Palabras clave](#)

Declaraciones de clase anidadas

Artículo • 03/03/2023 • Tiempo de lectura: 4 minutos

Una clase puede declararse dentro del ámbito de otra clase. Una clase de tipo se denomina "clase anidada". Las clases anidadas se consideran dentro del ámbito de la clase envolvente y están disponibles para su uso dentro de ese ámbito. Para hacer referencia a una clase anidada de un ámbito distinto al ámbito de inclusión inmediato, debe utilizar un nombre completo.

En el siguiente ejemplo se muestra cómo declarar clases anidadadas:

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` y `BufferedIO::BufferedOutput` se declaran en `BufferedIO`. Estos nombres de clase no son visibles fuera del ámbito de la clase `BufferedIO`. Sin embargo, un objeto de tipo `BufferedIO` no contiene objetos de los tipos `BufferedInput` o `BufferedOutput`.

Las clases anidadas pueden utilizar directamente nombres, nombres de tipo, nombres de miembros estáticos y enumeradores solo de la clase envolvente. Para usar nombres de otros miembros de clase, debe utilizar punteros, referencias o nombres de objeto.

En el ejemplo anterior de `BufferedIO`, pueden tener acceso directamente a la enumeración `IOError` funciones miembro de las clases anidadas,

`BufferedIO::BufferedInput` O `BufferedIO::BufferedOutput`, como se muestra en la función `good`.

ⓘ Nota

Las clases anidadas declaran solo tipos dentro del ámbito de la clase. No provocan la creación de objetos contenidos de la clase anidada. El ejemplo anterior declara dos clases anidadas pero no declara ningún objeto de estos tipos de clase.

Una excepción a la visibilidad del ámbito de una declaración de clase es cuando se declara un nombre de tipo junto con una declaración adelantada. En este caso, el nombre de clase declarado por la declaración adelantada está visible fuera de la clase envolvente, con el ámbito definido de modo que sea el menor envolvente no de clase. Por ejemplo:

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;
}
```

Privilegio de acceso en clases anidadas

El anidamiento de una clase dentro de otra clase no proporciona privilegios de acceso especiales a las funciones miembro de la clase anidada. De forma similar, las funciones miembro de la clase envolvente no tienen ningún acceso especial a los miembros de la clase anidada.

Funciones miembro en clases anidadas

Las funciones miembro declaradas en clases anidadas se pueden definir en el ámbito del archivo. El ejemplo anterior se podría haber escrito:

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()
{}
```

En el ejemplo anterior, se usa la sintaxis *qualified-type-name* para declarar el nombre de función. La declaración:

C++

```
BufferedIO::BufferedInput::read()
```

significa "la función `read` que es miembro de la clase `BufferedInput` que está en el ámbito de la clase `BufferedIO`". Dado que esta declaración usa la sintaxis *qualified-type-name*, se pueden crear construcciones de la siguiente forma:

C++

```
typedef BufferedIO::BufferedInput BIO_INPUT;  
  
int BIO_INPUT::read()
```

La declaración anterior es equivalente a la previa, pero usa un nombre de `typedef` en lugar de los nombres de clase.

Funciones friend en clases anidadas

Las funciones friend declaradas en una clase anidada se considera que están en el ámbito de la clase anidada, no la clase envolvente. Por lo tanto, las funciones friend no obtienen privilegios de acceso especiales a miembros o funciones miembro de la clase envolvente. Si desea utilizar un nombre declarado en una clase anidada en una función friend y la función friend está definida en el ámbito del archivo, debe usar nombres de tipo representativo del modo siguiente:

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()
{
}

```

El código siguiente muestra la función `GetExtendedErrorStatus` declarada como una función friend. En la función, que se define en el ámbito de archivo, se copia un mensaje de una matriz estática en un miembro de clase. Observe que una mejor implementación de `GetExtendedErrorStatus` consiste en declararlo como:

C++

```
int GetExtendedErrorStatus( char *message )
```

Con la interfaz anterior, varias clases pueden utilizar los servicios de esta función pasando una ubicación de memoria en la que desean que se copie el mensaje de error.

Consulte también

[Clases y structs](#)

Tipos de clase anónima

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Las clases pueden ser anónimas, es decir, se pueden declarar sin un *identificador*. Esto es útil cuando se reemplaza un nombre de clase por un nombre `typedef`, como en el siguiente ejemplo:

```
C++  
  
typedef struct  
{  
    unsigned x;  
    unsigned y;  
} POINT;
```

ⓘ Nota

El uso de las clases anónimas que se muestra en el ejemplo anterior es útil para mantener la compatibilidad con el código de C existente. En algún código de C, es frecuente el uso de `typedef` junto con estructuras anónimas.

Las clases anónimas también son útiles cuando se desea que una referencia a un miembro de clase aparezca como si no estuviera contenida en una clase independiente, como en el siguiente ejemplo de código:

```
C++  
  
struct PTValue  
{  
    POINT ptLoc;  
    union  
    {  
        int iValue;  
        long lValue;  
    };  
};  
  
PTValue ptv;
```

En el código anterior, se puede acceder a `iValue` usando el operador de selección de miembros de objeto (`.`) del modo siguiente:

```
C++
```

```
int i = ptv.iValue;
```

Las clases anónimas están sujetas a determinadas restricciones. (Para más información sobre las uniones anónimas, consulte [Uniones](#)). Clases anónimas:

- No pueden tener un constructor o un destructor.
- No se pueden pasar como argumentos a funciones (a menos que la comprobación de tipos se rechace mediante el uso de puntos suspensivos).
- No se pueden devolver como valores devueltos de funciones.

Structs anónimos

Específicos de Microsoft

Una extensión de Microsoft C permite declarar una variable de estructura dentro de otra estructura sin darle un nombre. Estas estructuras anidadas se denominan estructuras anónimas. C++ no permite estructuras anónimas.

Puede tener acceso a los miembros de una estructura anónima como si fueran miembros de la estructura contenedora.

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
```

FIN de Específicos de Microsoft

Punteros a miembros

Artículo • 03/03/2023 • Tiempo de lectura: 4 minutos

Las declaraciones de punteros a miembros son casos especiales de declaraciones de puntero. Se declaran mediante la secuencia siguiente:

```
storage-class-specifiersopt cv-qualifiersopt type-specifierms-modifieropt qualified-name :: * cv-qualifiersopt identifierpm-initializeropt ;
```

1. El especificador de declaración:

- Un especificador de clase de almacenamiento opcional.
- Especificadores `const` y `volatile` opcionales.
- El especificador de tipo: el nombre de un tipo. Es el tipo del miembro al que se apunta, no la clase.

2. El declarador:

- Modificador opcional concreto de Microsoft. Para obtener más información, consulte [Modificadores específicos de Microsoft](#).
- El nombre completo de la clase que contiene los miembros a los que se señala.
- El operador `::`.
- El operador `*`.
- Especificadores `const` y `volatile` opcionales.
- El identificador que denomina el puntero a miembro.

3. Un inicializador opcional de puntero a miembro:

- El operador `=`.
- El operador `&`.
- Nombre completo de la clase.
- El operador `::`.
- El nombre de un miembro no estático de la clase del tipo adecuado.

Como siempre, se permiten varios declaradores (y cualesquiera inicializadores asociados) en una sola declaración. Un puntero a un miembro podría no apuntar a un miembro estático de la clase, un miembro de tipo de referencia o `void`.

Un puntero a un miembro de una clase se diferencia de un puntero normal en que tiene información del tipo de miembro y de la clase a la que pertenece el miembro. Un puntero normal identifica (tiene la dirección de) un solo objeto en memoria. Un puntero a un miembro de una clase identifica ese miembro en cualquier instancia de la clase. En el ejemplo siguiente se declara una clase, `Window`, y algunos punteros a los datos de miembros.

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()
{
```

En el ejemplo anterior, `pwCaption` es un puntero a cualquier miembro de la clase `Window` que tenga el tipo `char*`. El tipo de `pwCaption` es `char * Window::*`. El siguiente fragmento de código declara punteros a las funciones miembro `SetCaption` y `GetCaption`.

C++

```
const char * (Window::* pfNWGC)() = &Window::GetCaption;
bool (Window::* pfNWSC)( const char * ) = &Window::SetCaption;
```

Los punteros `pfNWGC` y `pfNWSC` señalan, respectivamente, a `GetCaption` y a `SetCaption` de la clase `Window`. El código copia la información en la leyenda de la ventana directamente mediante el puntero al miembro `pwCaption`:

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';

```

La diferencia entre los operadores `.*` y `->*` (los operadores de puntero a miembro) consiste en que el operador `.*` selecciona los miembros con un objeto o una referencia de objeto especificados, mientras que el operador `->*` selecciona los miembros mediante un puntero. Para obtener más información sobre estos operadores, consulte [Expresiones con operadores de puntero a miembro](#).

El resultado de los operadores de puntero a miembro es el tipo de miembro. En este caso, es `char *`.

El fragmento de código siguiente invoca las funciones miembro `GetCaption` y `SetCaption` mediante punteros a miembros:

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 );

```

Restricciones de los punteros a miembros

La dirección de un miembro estático no es un puntero a un miembro. Es un puntero normal a la única instancia del miembro estático. Solo existe una instancia de un miembro estático para todos los objetos de una clase determinada. Esto significa que se pueden usar los operadores normales de dirección (`&`) y desreferencia (`*`).

Punteros a funciones virtuales y de miembro

La invocación de una función virtual mediante una función de puntero a miembro funciona como si se hubiera llamado directamente a la función. La función correcta se busca en v-table y se invoca.

La clave para trabajar con funciones virtuales es, como siempre, invocarlas a través de un puntero a una clase base. (Para obtener más información sobre las funciones virtuales, consulte [Funciones virtuales](#)).

El código siguiente muestra cómo invocar una función virtual a través de una función de puntero a miembro:

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
```

El puntero `this`.

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

El puntero `this` es un puntero accesible solo dentro de las funciones miembro no estáticas de un tipo `class`, `struct` o `union`. Señala al objeto para el que se llama a la función miembro. Las funciones miembro estáticas no tienen un puntero `this`.

Sintaxis

C++

```
this  
this->member-identifier
```

Comentarios

El puntero `this` de un objeto no forma parte del objeto en sí. No se refleja en el resultado de una instrucción `sizeof` en el objeto. Cuando se llama a una función miembro no estática para un objeto, el compilador pasa la dirección del objeto a la función como un argumento oculto. Por ejemplo, la siguiente llamada de función:

C++

```
myDate.setMonth( 3 );
```

se puede interpretar de esta manera:

C++

```
setMonth( &myDate, 3 );
```

La dirección del objeto está disponible desde la función miembro como puntero `this`. La mayoría de los usos del puntero `this` son implícitos. Es válido, aunque innecesario, usar explícitamente `this` cuando se hace referencia a los miembros de class. Por ejemplo:

C++

```
void Date::setMonth( int mn )  
{
```

```
month = mn;           // These three statements
this->month = mn;   // are equivalent
(*this).month = mn;
}
```

La expresión `*this` suele utilizarse para devolver el objeto actual desde una función miembro:

C++

```
return *this;
```

El puntero `this` también se usa para protegerse de la referencia automática:

C++

```
if (&Object != this) {
// do not execute in cases of self-reference
```

① Nota

Dado que el puntero `this` no es modificable, no se permiten asignaciones al puntero `this`. Las implementaciones anteriores de C++ permitían asignaciones a `this`.

En ocasiones, el puntero `this` se usa directamente; por ejemplo, para manipular elementos struct de datos a los que se hace referencia automáticamente, donde se requiere la dirección del objeto actual.

Ejemplo

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

Tipo del puntero `this`

El tipo del puntero `this` puede modificarse en la declaración de función mediante las palabras clave `const` y `volatile`. Para declarar una función que tenga cualquiera de estos atributos, agregue las palabras clave después de la lista de argumentos de función.

Considere un ejemplo:

C++

```
// type_of_this_pointer1.cpp
class Point
{
    unsigned X() const;
};
int main()
{}
```

El código anterior declara una función miembro, `X`, en la que el puntero `this` se trata como un puntero `const` a un objeto `const`. Aunque se pueden usar combinaciones de opciones *cv-mod-list*, siempre modifican el objeto al que apunta el puntero `this`, no el puntero en sí. La declaración siguiente declara la función `X`, donde el puntero `this` es un puntero `const` a un objeto `const`:

C++

```
// type_of_this_pointer2.cpp
class Point
{
    unsigned X() const;
};
int main()
{}
```

El tipo de `this` en una función miembro se describe mediante la sintaxis siguiente. El elemento *cv-qualifier-list* se determina a partir del declarador de la función

miembro. Puede ser `const` o `volatile` (o ambos). `class-type` es el nombre del elemento class:

[*cv-qualifier-list*] `class-type * const this`

En otras palabras, el puntero `this` siempre es un puntero `const`. No se puede reasignar. Los calificadores `const` o `volatile` que se usan en la declaración de funciones miembro se aplican a la instancia class a la que apunta el puntero `this` en el ámbito de dicha función.

En la tabla siguiente se explica más detalladamente el funcionamiento de estos modificadores.

Semántica de modificadores `this`

| Modificador | Significado |
|-----------------------|---|
| <code>const</code> | No se pueden cambiar los datos de los miembros; no se pueden invocar funciones miembro que no sean <code>const</code> . |
| <code>volatile</code> | Los datos de miembros se cargan desde la memoria cada vez que se obtiene acceso; deshabilita ciertas optimizaciones. |

Es un error pasar un objeto `const` a una función miembro que no sea `const`.

De igual forma, es un error pasar un objeto `volatile` a una función miembro que no sea `volatile`.

Las funciones miembro declaradas como `const` no pueden cambiar los datos de miembro (en dichas funciones, el puntero `this` es un puntero a un objeto `const`).

ⓘ Nota

Los constructores y desestructores no se pueden declarar como `const` o `volatile`. Pueden, sin embargo, invocarse en objetos `const` o `volatile`.

Consulte también

[Palabras clave](#)

Campos de bits de C++

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Las clases y estructuras pueden contener miembros que ocupan menos almacenamiento que un tipo entero. Estos miembros se especifican como campos de bits. La sintaxis para la especificación de *member-declarator* de un campo de bits es la siguiente:

Sintaxis

declarador:constant-expression

Comentarios

El *declarador* (opcional) es el nombre por el que se tiene acceso al miembro en el programa. Debe ser de tipo entero (incluidos los tipos enumerados). *constant-expression* especifica el número de bits que ocupa el miembro en la estructura. Se pueden utilizar campos de bits anónimos (es decir, miembros de campos de bits sin identificador) para llenar.

ⓘ Nota

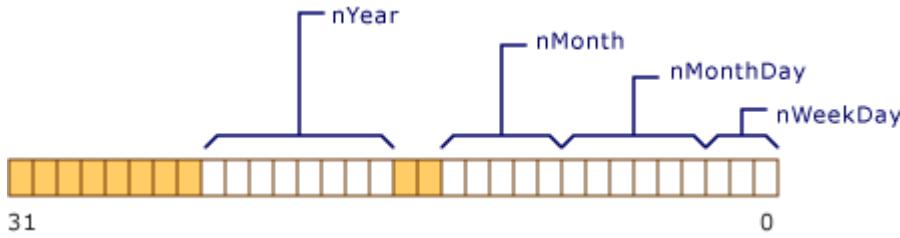
Un campo de bits sin nombre de ancho 0 fuerza la alineación del siguiente campo de bits con el siguiente **tipo** de límite, donde **tipo** es el tipo del miembro.

En el ejemplo siguiente se declara una estructura que contiene campos de bits:

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)
};
```

En la ilustración siguiente se muestra el diseño de memoria conceptual de un objeto de tipo `Date`.



Diseño de memoria de objeto de fecha

Tenga en cuenta que `nYear` tiene 8 bits de longitud y desbordaría el límite de palabra del tipo declarado, `unsigned short`. Por lo tanto, se inicia al principio de una nueva `unsigned short`. No es necesario que todos los campos de bits quepan en un objeto del tipo subyacente; se asignan nuevas unidades de almacenamiento según el número de bits solicitados en la declaración.

Específicos de Microsoft

El orden de los datos declarados como campos de bits es de bit bajo a bit alto, como se muestra en la ilustración anterior.

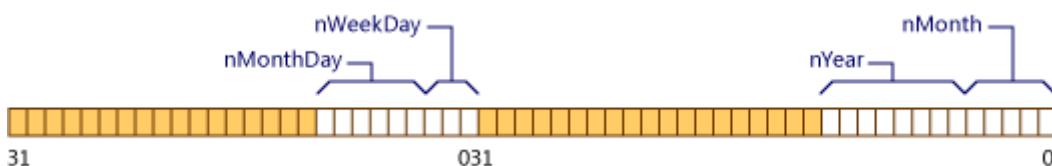
FIN de Específicos de Microsoft

Si la declaración de una estructura incluye un campo sin nombre de longitud 0, como se muestra en el ejemplo siguiente,

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)
};
```

a continuación, el diseño de memoria es como se muestra en la ilustración siguiente:



Diseño de objeto de fecha con campo de bits de longitud cero

El tipo subyacente de un campo de bits debe ser un tipo entero, como se describe en [tipos integrados](#).

Si el inicializador de una referencia de tipo `const T&` es un valor lvalue que hace referencia a un campo de bits de tipo `T`, la referencia no está enlazada al campo de bits directamente. En su lugar, la referencia se enlaza a un inicializado temporalmente para contener el valor del campo de bits.

Restricciones de los campos de bits

En la lista siguiente se detallan operaciones erróneas en campos de bits:

- Tomar la dirección de un campo de bits.
- Inicializar una no `const` referencia con un campo de bits.

Consulte también

[Clases y structs](#)

Expresiones lambda en C++

Artículo • 03/03/2023 • Tiempo de lectura: 13 minutos

En C++11 y versiones posteriores, una expresión lambda, a menudo denominada *lambda*, es una manera cómoda de definir un objeto de función anónimo (un *cierre*) justo en la ubicación donde se invoca o se pasa como argumento a una función. Normalmente, las expresiones lambda se usan para encapsular unas líneas de código que se pasan a algoritmos o métodos asincrónicos. En este artículo se definen las expresiones lambda y se comparan con otras técnicas de programación. Describe sus ventajas y proporciona algunos ejemplos básicos.

Artículos relacionados

- [Expresiones lambda frente a objetos de función](#)
- [Trabajar con expresiones lambda](#)
- [Expresiones lambda constexpr](#)

Partes de una expresión lambda

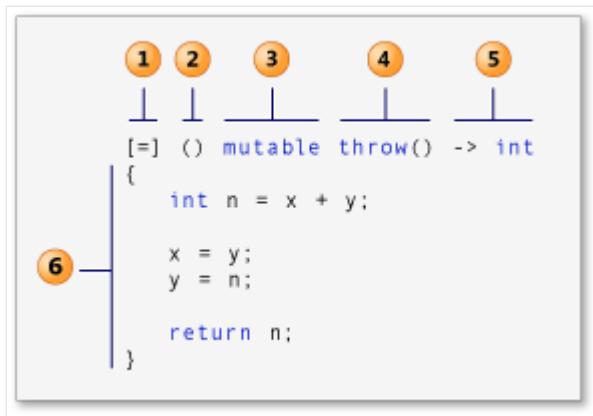
Esta es una expresión lambda simple que se pasa como tercer argumento a la `std::sort()` función:

```
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
    );
}
```

En esta ilustración se muestran las partes de la sintaxis lambda:



1. *cláusula de captura* (también conocida como *iniciador de expresión lambda* en la especificación de C++)
2. *lista de parámetros Opcional*. (también conocida como *declarador de expresión lambda*)
3. *especificación mutable Opcional*.
4. *especificación de excepción Opcional*.
5. *tipo de valor devuelto final Opcional*.
6. *cuerpo lambda*.

Cláusula de captura

Una expresión lambda puede introducir nuevas variables en su cuerpo (en C++14) y también puede acceder a variables o *capturarlas* del ámbito circundante. Una expresión lambda comienza con la cláusula de captura. Especifica qué variables se capturan y si la captura es por valor o por referencia. A las variables que tienen como prefijo y comercial (&) se accede mediante referencia y a las variables que no tienen el prefijo se accede por valor.

Una cláusula de captura vacía, `[]`, indica que el cuerpo de la expresión lambda no tiene acceso a ninguna variable en el ámbito de inclusión.

Puede usar un modo de captura predeterminado para indicar cómo capturar las variables externas a las que se hace referencia en el cuerpo lambda: `[&]` significa que todas las variables a las que hace referencia se capturan por referencia y `[=]` significa que se capturan por valor. Puede usar un modo de captura predeterminado y, después, especificar el modo opuesto de forma explícita para unas variables específicas. Por ejemplo, si el cuerpo de una expresión lambda accede a la variable externa `total` por referencia y a la variable externa `factor` por valor, las siguientes cláusulas capture serán equivalentes:

C++

```
[&total, factor]
[factor, &total]
[&, factor]
[=, &total]
```

Solo se capturan las variables que se mencionan en el cuerpo lambda cuando se usa capture-default.

Si una cláusula de captura incluye una cláusula capture-default `&`, ningún identificador de una captura de esa cláusula de captura puede tener el formato `&identifier`. Del mismo modo, si la cláusula de captura incluye una cláusula capture-default `=`, ninguna captura de esa cláusula de captura puede tener el formato `=identifier`. Un identificador o `this` no puede aparecer más de una vez en una cláusula de captura. En el fragmento de código siguiente se muestran algunos ejemplos:

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
}
```

Una captura seguida de puntos suspensivos es una expansión de paquetes, como se muestra en este ejemplo de: [plantilla variádica](#)

C++

```
template<class... Args>
void f(Args... args) {
    auto x = [args...] { return g(args...); };
    x();
}
```

Para usar expresiones lambda en el cuerpo de una función miembro de clase, pase el puntero `this` a la cláusula de captura para proporcionar acceso a las funciones miembro y a los miembros de datos de la clase contenedora.

Visual Studio 2017 versión 15.3 y posteriores (disponible en modo `/std:c++17` y versiones posteriores): el puntero `this` se puede capturar por valor especificando `*this` en la cláusula de captura. La captura por valor copia el cierre completo en cada sitio de llamada donde se invoca la expresión lambda. (Un cierre es el objeto de función anónima que encapsula la expresión lambda). La captura por valor es útil cuando la expresión lambda se ejecuta en operaciones paralelas o asincrónicas. Es especialmente útil en determinadas arquitecturas de hardware, como NUMA.

Para obtener un ejemplo en el que se muestra cómo usar expresiones lambda con funciones miembro de clase, vea "Ejemplo: Uso de una expresión lambda en un método" en [Ejemplos de expresiones lambda](#).

Al usar la cláusula capture, se recomienda tener en cuenta estos puntos, especialmente cuando se usan expresiones lambda con multiprocesamiento:

- Se pueden usar capturas por referencia para modificar variables externas, pero no se pueden usar capturas por valor. (`mutable` permite modificar copias, pero no originales)
- Las capturas por referencia reflejan actualizaciones en variables externas, pero las capturas por valor no.
- Las capturas por referencia presentan una dependencia de la duración, pero las capturas por valor no tienen ninguna dependencia de la duración. Esto es muy importante cuando la expresión lambda se inicia de forma asincrónica. Si captura un valor local por referencia en una expresión lambda asincrónica, esa configuración local podría haber desaparecido fácilmente en el momento en que se ejecuta la expresión lambda. El código podría provocar una infracción de acceso en tiempo de ejecución.

Captura generalizada (C++ 14)

En C++14, puede introducir e inicializar nuevas variables en la cláusula de captura, sin necesidad de que esas variables existan en el ámbito envolvente de la función lambda. La inicialización se puede expresar como cualquier expresión arbitraria; el tipo de la variable nueva se deduce del tipo producido por la expresión. Esta característica le permite capturar variables de solo movimiento (como `std::unique_ptr`) del ámbito circundante y usarlas en una expresión lambda.

C++

```
pNums = make_unique<vector<int>>(nums);  
//...
```

```
auto a = [ptr = move(pNums)]()
{
    // use ptr
};
```

Lista de parámetros

Las expresiones lambda pueden capturar variables y aceptar parámetros de entrada. La lista de parámetros (*declarador de expresión lambda* en la sintaxis estándar) es opcional y, en la mayoría de los aspectos, es similar a la lista de parámetros de una función.

C++

```
auto y = [] (int first, int second)
{
    return first + second;
};
```

En C++14, si el tipo de parámetro es genérico, puede usar la palabra clave `auto` como especificador de tipo. Esto indica al compilador que debe crear el operador de llamada de función como plantilla. Cada instancia de `auto` en una lista de parámetros es equivalente a un parámetro de tipo distinto.

C++

```
auto y = [] (auto first, auto second)
{
    return first + second;
};
```

Una expresión lambda puede tomar otra expresión lambda como argumento. Para obtener más información, vea "Expresiones lambda de orden superior" en el artículo [ejemplos de expresiones lambda](#).

Dado que una lista de parámetros es opcional, puede omitir los paréntesis vacíos si no pasa argumentos a la expresión lambda y su declarador lambda no contiene una *especificación de excepción*, *trailing-return-type* o `mutable`.

Especificación mutable

Normalmente, el operador de llamada de función de una expresión lambda es `const`-`by-value`, pero el uso de la palabra clave `mutable` lo cancela. No genera miembros de datos mutables. La especificación `mutable` permite al cuerpo de una expresión lambda

modificar las variables que se capturan por valor. Algunos de los ejemplos que se incluyen más adelante en este artículo muestran el uso de `mutable`.

Especificación de la excepción

Puede usar la especificación de excepción `noexcept` para indicar que la expresión lambda no produce ninguna excepción. Al igual que con las funciones normales, el compilador de Microsoft C++ genera una advertencia [C4297](#) si una expresión lambda declara la especificación de excepción `noexcept` y el cuerpo lambda produce una excepción, como se muestra aquí:

C++

```
// throw_lambda_expression.cpp
// compile with: /W4 /EHsc
int main() // C4297 expected
{
    []() noexcept { throw 5; }();
}
```

Para obtener más información, vea [Especificaciones de excepción \(throw\)](#).

Tipo de valor devuelto

El tipo de valor devuelto de una expresión lambda se deduce automáticamente. No tiene que usar la palabra clave `auto` a menos que especifique un *trailing-return-type*. El *trailing-return-type* es similar a la parte de tipo de valor devuelto de una función normal o una función miembro. Pero el tipo de valor devuelto debe seguir la lista de parámetros y debe incluir la palabra clave trailing-return-type `->` antes del tipo de valor devuelto.

Puede omitir la parte de tipo de valor devuelto de una expresión lambda si el cuerpo lambda contiene solo una instrucción return. O bien, si la expresión no devuelve un valor. Si el cuerpo de la expresión lambda contiene una instrucción return, el compilador deduce el tipo de valor devuelto del tipo de expresión return. En caso contrario, el compilador deduce que el tipo de valor devuelto es `void`. Vea los fragmentos de código de ejemplo siguientes que muestran este principio:

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
```

Una expresión lambda puede generar otra expresión lambda como valor devuelto. Para obtener más información, vea "Expresiones lambda de orden superior" en [Ejemplos de expresiones lambda](#).

Cuerpo lambda

El cuerpo lambda de una expresión lambda es una instrucción compuesta. Puede contener todo lo que se permite en el cuerpo de una función normal o una función miembro. El cuerpo de una función normal y de una expresión lambda puede tener acceso a estos tipos de variables:

- variables capturadas en el ámbito de inclusión, tal como se describió anteriormente.
- Parámetros.
- Variables declaradas localmente.
- Miembros de datos de la clase, cuando se declara dentro de una clase y se captura `this`.
- Cualquier variable que tenga duración de almacenamiento estática, por ejemplo, variables globales.

El ejemplo siguiente contiene una expresión lambda que captura explícitamente la variable `n` por valor y captura implícitamente la variable `m` por referencia:

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

Como la variable `n` se captura por valor, el valor sigue siendo `0` después de la llamada a la expresión lambda. La especificación `mutable` permite modificar `n` dentro de la expresión lambda.

Una expresión lambda solo puede capturar variables que tengan una duración de almacenamiento automática. Sin embargo, puede usar variables que tengan una duración de almacenamiento estática en el cuerpo de una expresión lambda. En el ejemplo siguiente se utiliza la función `generate` y una expresión lambda para asignar un valor a cada elemento de un objeto `vector`. La expresión lambda modifica la variable estática para generar el valor del elemento siguiente.

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
}
```

Para obtener más información, vea [generar](#).

En el ejemplo de código siguiente se usa la función del ejemplo anterior y se agrega un ejemplo de una expresión lambda que usa el algoritmo de biblioteca estándar C++ `generate_n`. Esta expresión lambda asigna un elemento de un objeto `vector` a la suma de los dos elementos anteriores. Se usa la palabra clave `mutable` de modo que el cuerpo de la expresión lambda pueda modificar sus copias de las variables externas `x` e `y`, que la expresión lambda captura por valor. Como la expresión lambda captura las variables originales `x` e `y` por valor, sus valores siguen siendo `1` después de la ejecución de la expresión.

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

```

Para obtener más información, vea [generate_n](#).

constexpr expresiones lambda

Visual Studio 2017 versión 15.3 y posteriores (disponible en modo `/std:c++17` y versiones posteriores): puede declarar una expresión lambda como `constexpr` (o usarla en una expresión constante) cuando se permite la inicialización de cada miembro de datos capturado o introducido dentro de una expresión constante.

C++

```

int y = 32;
auto answer = [y]() constexpr
{
    int x = 10;
    return y + x;
};

constexpr int Increment(int n)
{
    return [n] { return n + 1; }();
}

```

Una expresión lambda es implícitamente `constexpr` si su resultado cumple los requisitos de una función `constexpr`:

C++

```

auto answer = [](int n)
{

```

```
    return 32 + n;
};

constexpr int response = answer(10);
```

Si una expresión lambda es implícita o explícitamente `constexpr`, la conversión a un puntero de función genera una función `constexpr`:

C++

```
auto Increment = [](int n)
{
    return n + 1;
};

constexpr int(*inc)(int) = Increment;
```

Específico de Microsoft

Las expresiones lambda no se admite en las entidades administradas siguientes de Common Language Runtime (CLR): `ref class`, `ref struct`, `value class`, o `value struct`.

Si usa un modificador específico de Microsoft como `_declspec`, puede insertarlo en una expresión lambda inmediatamente después de la `parameter-declaration-clause`. Por ejemplo:

C++

```
auto Sqr = [](int t) __declspec(code_seg("PagedMem")) -> int { return t*t;
};
```

Para determinar si un modificador es compatible con las expresiones lambda, consulte el artículo correspondiente en la sección [Modificadores específicos de Microsoft de la documentación](#).

Visual Studio admite la funcionalidad lambda estándar de C++11 y *lambda sin estado*. Una expresión lambda sin estado se puede convertir en un puntero de función que usa una convención de llamada arbitraria.

Vea también

[Referencia del lenguaje C++](#)

[Objetos de función en la biblioteca estándar de C++](#)

Llamada a función

for_each

Sintaxis de la expresión lambda

Artículo • 29/09/2022 • Tiempo de lectura: 4 minutos

En este artículo se demuestra la sintaxis y los elementos estructurales de las expresiones lambda. Para ver una descripción de las expresiones lambda, consulte [Expresiones lambda](#).

Objetos de función frente a expresiones lambda

Al escribir código, probablemente use punteros de función y objetos de función para resolver problemas y realizar cálculos, especialmente cuando se usan [algoritmos de la biblioteca estándar de C++](#). Tanto los punteros a función como los objetos de función tienen ventajas y desventajas; por ejemplo, los punteros a función tienen una sobrecarga sintáctica mínima pero no conservan el estado dentro de un ámbito, y los objetos de función pueden conservar el estado pero requieren la sobrecarga sintáctica de una definición de clase.

Una expresión lambda combina las ventajas de los punteros a función y los objetos de función y evita sus desventajas. Al igual que los objetos de función, una expresión lambda es flexible y puede conservar el estado, pero, a diferencia de un objeto de función, su sintaxis compacta no requiere una definición de clase explícita. Mediante expresiones lambda, se puede escribir código menos complejo y menos propenso a errores que el código para un objeto de función equivalente.

En los ejemplos siguientes se compara el uso de una expresión lambda con el uso de un objeto de función. En el primer ejemplo se utiliza una expresión lambda para imprimir en la consola si cada elemento de un objeto `vector` es par o impar. En el segundo ejemplo se usa un objeto de función para realizar la misma tarea.

Ejemplo 1: utilizar una expresión lambda

En este ejemplo se pasa una expresión lambda a la función `for_each`. La expresión lambda imprime un resultado que indica si cada elemento de un objeto `vector` es par o impar.

Código

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.

```

Comentarios

En el ejemplo, el tercer argumento a la función `for_each` es una expresión lambda. La parte `[&evenCount]` especifica la cláusula capture de la expresión, `(int n)` especifica la

lista de parámetros y la parte restante especifica el cuerpo de la expresión.

Ejemplo 2: utilizar un objeto de función

En ocasiones, una expresión lambda sería demasiado difícil de extender mucho más allá del ejemplo anterior. En el ejemplo siguiente se utiliza un objeto de función en lugar de una expresión lambda, junto con la función `for_each`, para generar los mismos resultados que en el ejemplo 1. Ambos ejemplos almacenan el recuento de números pares en un objeto `vector`. Para mantener el estado de la operación, la clase

`FunctorClass` almacena la variable `m_evenCount` por referencia como una variable miembro. Para realizar la operación, `FunctorClass` implementa el operador de llamada a función, `operator()`. El compilador de Microsoft C++ genera código que es comparable en cuanto a tamaño y rendimiento con el código lambda del ejemplo 1. Si se trata de un problema básico como el de este artículo, probablemente el diseño lambda más simple sea mejor que el diseño de objeto de función. Sin embargo, si cree que la funcionalidad puede requerir una extensión importante en el futuro, utilice el diseño de objeto de función para que el mantenimiento del código sea más sencillo.

Para más información sobre `operator()`, consulte [Llamada a función](#). Para más información sobre la función `for_each`, consulte [for_each](#).

Código

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.

```

Consulte también

[Expresiones lambda](#)

[Ejemplos de expresiones lambda](#)

[generate](#)

[generate_n](#)

[for_each](#)

[Especificaciones de excepciones \(throw\)](#)

[Advertencia del compilador \(nivel 1\) C4297](#)

[Modificadores específicos de Microsoft](#)

Ejemplos de expresiones lambda

Artículo • 03/03/2023 • Tiempo de lectura: 10 minutos

En este artículo se muestra cómo usar expresiones lambda en programas. Para ver una introducción a las expresiones lambda, consulte [Expresiones lambda](#). Para más información sobre la estructura de una expresión lambda, consulte [Sintaxis de las expresiones lambda](#).

Declarar expresiones lambda

Ejemplo 1

Puesto que una expresión lambda tiene tipo, puede asignarla a una variable `auto` o a un objeto `function`, como se muestra aquí:

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;
}
```

El ejemplo produce la siguiente salida:

Output

```
5
7
```

Comentarios

Para más información, consulte [auto](#), [Clase function](#) y [Llamada a función](#).

Aunque las expresiones lambda se suelen declarar en el cuerpo de una función, se pueden declarar en cualquier lugar donde se pueda inicializar una variable.

Ejemplo 2

El compilador de Microsoft C++ enlaza una expresión lambda a sus variables capturadas cuando se declara la expresión, no cuando se llama a la misma. En el ejemplo siguiente se muestra una expresión lambda que captura la variable local `i` por valor y la variable local `j` por referencia. Como la expresión lambda captura `i` por valor, la reasignación de `i` más adelante en el programa no afecta al resultado de la expresión. Sin embargo, puesto que la expresión lambda captura `j` por referencia, la reasignación de `j` afecta al resultado de la expresión.

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;
}
```

El ejemplo produce la siguiente salida:

Output

[En este artículo]

Llamar a expresiones lambda

Es posible llamar a una expresión lambda inmediatamente, como se muestra en el fragmento de código siguiente. En el segundo fragmento de código se muestra cómo pasar una expresión lambda como argumento a algoritmos de la biblioteca estándar de C++ como `find_if`.

Ejemplo 1

En este ejemplo se declara una expresión lambda que devuelve la suma de dos números enteros y se llama a la expresión inmediatamente con los argumentos 5 y 4:

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;
}
```

El ejemplo produce la siguiente salida:

Output

```
9
```

Ejemplo 2

En este ejemplo se pasa una expresión lambda como argumento a la función `find_if`. La expresión lambda devuelve `true` si el parámetro es un número par.

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;
    }
}

```

El ejemplo produce la siguiente salida:

Output

The first even number in the list is 42.

Comentarios

Para más información sobre la función `find_if`, consulte [find_if](#). Para más información sobre las funciones de la biblioteca estándar de C++ que realizan los algoritmos comunes, consulte [<algorithm>](#).

[\[En este artículo\]](#)

Anidar expresiones lambda

Ejemplo

Se puede anidar una expresión lambda dentro de otra, como se muestra en este ejemplo. La expresión lambda interna multiplica su argumento por 2 y devuelve el resultado. La expresión lambda externa llama a la expresión lambda interna con su argumento y suma 3 al resultado.

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;
}
```

El ejemplo produce la siguiente salida:

Output

13

Comentarios

En este ejemplo, `[](int y) { return y * 2; }` es la expresión lambda anidada.

[[En este artículo](#)]

Funciones lambda de orden superior

Ejemplo

Muchos lenguajes de programación admiten el concepto de *una función de orden superior*. Una función de orden superior es una expresión lambda que toma otra expresión lambda como argumento o devuelve una expresión lambda. Se puede usar la clase `function` para permitir que una expresión lambda de C++ se comporte como una función de orden superior. En el ejemplo siguiente se muestra una expresión lambda que devuelve un objeto `function` y una expresión lambda que toma un objeto `function` como argumento.

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;
}
```

El ejemplo produce la siguiente salida:

Output

30

[\[En este artículo\]](#)

Usar una expresión lambda en una función

Ejemplo

Las expresiones lambda se pueden usar en el cuerpo de una función. La expresión lambda puede tener acceso a cualquier función o miembro de datos al que pueda tener acceso la función envolvente. Se puede capturar explícita o implícitamente el puntero `this` para proporcionar acceso a las funciones y miembros de datos de la clase envolvente. **Visual Studio 2017 versión 15.3 y posteriores** (disponible con `/std:c++17` y versiones posteriores): capture `this` por valor (`[*this]`) cuando la expresión lambda se use en operaciones asincrónicas o paralelas en las que el código podría ejecutarse después de que el objeto original salga del ámbito.

Se puede usar el puntero `this` explícitamente en una función, como se muestra aquí:

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; });
}
```

También se puede capturar el puntero `this` de forma implícita:

C++

```
void ApplyScale(const vector<int>& v) const
{
    for_each(v.begin(), v.end(),
        [=](int n) { cout << n * _scale << endl; });
}
```

En el ejemplo siguiente se muestra la clase `Scale`, que encapsula un valor de escala.

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. Does not modify the vector.
    Scale s(3);
    s.ApplyScale(values);
}

```

El ejemplo produce la siguiente salida:

Output

```

3
6
9
12

```

Comentarios

La función `ApplyScale` usa una expresión lambda para imprimir el producto del valor de escala y cada elemento de un objeto `vector`. La expresión lambda captura implícitamente el puntero `this` para poder tener acceso al miembro `_scale`.

[En este artículo]

Usar expresiones lambda con plantillas

Ejemplo

Puesto que las expresiones lambda tienen tipo, pueden utilizarse con plantillas de C++. En el ejemplo siguiente se muestran las funciones `negate_all` y `print_all`. La función `negate_all` aplica el `operator-` unario a todos los elementos del objeto `vector`. La función `print_all` imprime en la consola todos los elementos del objeto `vector`.

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);
}
```

El ejemplo produce la siguiente salida:

Output

```
34
-43
56
After negate_all():
-34
43
-56
```

Comentarios

Para más información sobre las plantillas de C++, consulte [Plantillas](#).

[[En este artículo](#)]

Controlar las excepciones

Ejemplo

El cuerpo de una expresión lambda sigue las reglas tanto del control de excepciones estructurado (SEH) como del control de excepciones de C++. Es posible controlar una excepción generada en el cuerpo de una expresión lambda o aplazar el control de excepciones al ámbito de inclusión. En el ejemplo siguiente se usa la función `for_each` y una expresión lambda para llenar un objeto `vector` con los valores de otro. Se utiliza un bloque `try/catch` para controlar el acceso no válido al primer vector.

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;
};
}

```

El ejemplo produce la siguiente salida:

Output

Caught 'invalid vector<T> subscript'.

Comentarios

Para más información sobre el control de excepciones, consulte [Control de excepciones](#).

[\[En este artículo\]](#)

Usar expresiones lambda con tipos administrados (C++/CLI)

Ejemplo

La cláusula capture de una expresión lambda no puede contener una variable que tenga un tipo administrado. Sin embargo, se puede pasar un argumento que tenga un tipo administrado a la lista de parámetros de una expresión lambda. El ejemplo siguiente

contiene una expresión lambda que captura la variable local no administrada `ch` por valor y toma un objeto `System.String` como parámetro.

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");
}
```

El ejemplo produce la siguiente salida:

Output

```
Hello!
```

Comentarios

También se pueden usar expresiones lambda con la biblioteca de STL/CLR. Para más información, consulte [Referencia de la biblioteca STL/CLR](#).

Importante

Las expresiones lambda no se admite en estas entidades administradas siguientes de Common Language Runtime (CLR): `ref class`, `ref struct`, `value class` y `value struct`.

[[En este artículo](#)]

Consulte también

[Expresiones lambda](#)

[Sintaxis de la expresión lambda](#)

auto

Clase function

find_if

<algorithm>

Llamada a función

Templates (Plantillas [C++])

Control de excepciones

Referencia de la biblioteca STL/CLR

Expresiones lambda de `constexpr` en C++

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Visual Studio 2017 versión 15.3 y posteriores (disponible en modo `/std:c++17` y versiones posteriores): una expresión lambda se puede declarar como `constexpr` o usarse en una expresión constante cuando la inicialización de cada miembro de datos que captura o introduce se permite dentro de una expresión constante.

C++

```
int y = 32;
auto answer = [y]() constexpr
{
    int x = 10;
    return y + x;
};

constexpr int Increment(int n)
{
    return [n] { return n + 1; }();
```

Una expresión lambda es implícitamente `constexpr` si su resultado cumple los requisitos de una función `constexpr`:

C++

```
auto answer = [](int n)
{
    return 32 + n;
};

constexpr int response = answer(10);
```

Si una expresión lambda es implícita o explícitamente `constexpr` y la convierte en un puntero de función, la función resultante también es `constexpr`:

C++

```
auto Increment = [](int n)
{
    return n + 1;
};
```

```
constexpr int(*inc)(int) = Increment;
```

Vea también

[Referencia del lenguaje C++](#)

[Objetos de función en la biblioteca estándar de C++](#)

[Llamada a función](#)

[for_each](#)

Matrices (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 12 minutos

Una matriz es una secuencia de objetos del mismo tipo que ocupan un área contigua de memoria. Las matrices tradicionales de estilo C son el origen de muchos errores, pero siguen siendo comunes, especialmente en las bases de código anteriores. En C++moderno, se recomienda encarecidamente usar `std::vector` o `std::array` en lugar de matrices de estilo C descritas en esta sección. Ambos de estos tipos de biblioteca estándar almacenan sus elementos como un bloque contiguo de memoria. Sin embargo, proporcionan mayor seguridad de tipos y admiten iteradores que se garantizan para que apunten a una ubicación válida dentro de la secuencia. Para obtener más información, consulte [Containers](#).

Declaraciones de pila

En una declaración de matriz de C++, el tamaño de la matriz se especifica después del nombre de la variable, no después del nombre de tipo como en otros lenguajes. El siguiente ejemplo declara una matriz de 1000 dobles que se asignarán en la pila. El número de elementos debe proporcionarse como un literal entero o como una expresión constante. Esto se debe a que el compilador tiene que saber cuánto espacio de pila asignar; no puede usar un valor calculado en tiempo de ejecución. A cada elemento de la matriz se le asigna un valor predeterminado de 0. Si no se asigna un valor predeterminado, cada elemento contiene inicialmente cualquier valor aleatorio que se encuentre en esa ubicación de memoria.

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] << " "  
}
```

El primer elemento de la matriz es el elemento zeroth. El último elemento es el elemento ($n-1$), donde n es el número de elementos que la matriz puede contener. El número de elementos de la declaración debe ser de un tipo entero y debe ser mayor que 0. Es su responsabilidad asegurarse de que el programa nunca pasa un valor al operador de subíndice mayor que `(size - 1)`.

Una matriz de tamaño cero es válida únicamente cuando la matriz es el último campo en un `struct` o `union` y cuando las extensiones de Microsoft están habilitadas (`/za` o `/permissive-` no está establecido).

Las matrices basadas en pila son más rápidas para asignar y acceder a las matrices basadas en montón. Sin embargo, el espacio de pila es limitado. El número de elementos de matriz no puede ser tan grande que use demasiada memoria de pila. Cuánto es demasiado depende de su programa. Puede usar herramientas de generación de perfiles para determinar si una matriz es demasiado grande.

Declaraciones abiertas

Puede requerir una matriz demasiado grande para asignar en la pila o cuyo tamaño no se conoce en tiempo de compilación. Es posible asignar esta matriz en el montón mediante una expresión `new[]`. El operador devuelve un puntero al primer elemento. El operador de subíndice funciona en la variable de puntero de la misma manera que en una matriz basada en pila. También puede usar la [aritmética de puntero](#) para mover el puntero a cualquier elemento arbitrario de la matriz. Es su responsabilidad asegurarse de ello:

- siempre mantiene una copia de la dirección del puntero original para que pueda eliminar la memoria cuando ya no necesite la matriz.
- no incrementa o disminuye la dirección del puntero más allá de los límites de la matriz.

En el ejemplo siguiente se muestra cómo definir una matriz en el montón en tiempo de ejecución. Muestra cómo obtener acceso a los elementos de matriz mediante el operador de subíndice y mediante la aritmética de punteros:

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);
}

```

Inicializar matrices

Se puede inicializar una matriz en un bucle, un elemento a la vez o en una sola instrucción. El contenido de las dos matrices siguientes es idéntico:

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 };
```

Pasar matrices a funciones

Cuando se pasa una matriz a una función, se pasa como puntero al primer elemento, ya sea una matriz basada en pila o en montón. El puntero no contiene ninguna otra información de tamaño o tipo. Este comportamiento se denomina *degradación del puntero*. Al pasar una matriz a una función, siempre se debe especificar el número de elementos en un parámetro independiente. Este comportamiento también implica que los elementos de matriz no se copian cuando la matriz se pasa a una función. Para evitar que la función modifique los elementos, especifique el parámetro como puntero a los elementos `const`.

En el ejemplo siguiente se muestra una función que acepta una matriz y una longitud. El puntero apunta a la matriz original y no a una copia. Dado que el parámetro no es `const`, la función puede modificar los elementos de matriz.

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]
    }
}
```

Declare y defina el parámetro de matriz `p` como `const` para que sea de solo lectura dentro del bloque de función:

C++

```
void process(const double *p, const size_t len);
```

La misma función también se puede declarar de estas maneras, sin cambios en el comportamiento. La matriz se sigue pasando como puntero al primer elemento:

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);
```

Matrices multidimensionales

Las matrices construidas a partir de otras matrices son matrices multidimensionales. Estas matrices multidimensionales se especifican colocando en orden varias expresiones constantes entre corchetes. Por ejemplo, considere esta declaración:

C++

```
int i2[5][7];
```

Especifica una matriz de tipo `int`, organizada conceptualmente en una matriz bidimensional de cinco filas y siete columnas, como se muestra en la ilustración siguiente:

| | | | | | | |
|------|------|------|------|------|------|------|
| 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 |

La imagen es una cuadrícula de 7 celdas de ancho y 5 celdas altas. Cada celda contiene el índice de la celda. El primer índice de celda tiene la etiqueta 0,0. La siguiente celda de esa fila es 0,1, etc. hasta la última celda de esa fila, que es 0,6. La siguiente fila comienza con el índice 1,0. La celda después de eso tiene un índice de 1,1. La última celda de esa fila es 1,6. Este patrón se repite hasta la última fila, que comienza con el índice 4,0. La última celda de la última fila tiene un índice de 4,6. :::image-end

Puede declarar matrices multidimensionales que tienen una lista de inicializadores (como se describe en [Inicializadores](#)). En estas declaraciones se puede omitir la expresión constante que especifica los límites de la primera dimensión. Por ejemplo:

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 }
};

```

La declaración anterior define una matriz de tres filas por cuatro columnas. Las filas representan fábricas y las columnas representan los mercados a los que distribuyen las fábricas. Los valores son los costos de transporte de las fábricas a los mercados. La primera dimensión de la matriz se omite, pero el compilador la completa examinando el inicializador.

El uso del operador de direccionamiento indirecto (*) en un tipo de matriz de n dimensiones produce una matriz de n-1 dimensiones. Si n es 1, el resultado es un valor escalar (o elemento de matriz).

Las matrices de C++ se almacenan por orden de fila principal. El orden de fila principal significa que el último subíndice es el que varía más rápidamente.

Ejemplo

También se puede omitir la especificación de límites de la primera dimensión de una matriz multidimensional en declaraciones de función, como se muestra aquí:

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

La función `FindMinToMkt` se escribe de forma que para agregar nuevas fábricas no sea necesario modificar el código, solo volver a compilarlo.

Inicializar matrices

El constructor inicializa las matrices de objetos que tienen un constructor de clase. Cuando hay menos elementos en la lista de inicializadores que en la matriz, se usa el constructor predeterminado para los elementos restantes. Si no se define ningún constructor predeterminado para la clase, la lista de inicializadores debe estar *completa*, es decir, debe haber un inicializador para cada elemento de la matriz.

Considere la clase `Point`, que define dos constructores:

```

// 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()
{
}

```

El primer elemento de `aPoint` se construye utilizando el constructor `Point(int, int)`; los dos elementos restantes se crean con el constructor predeterminado.

Las matrices de miembros estáticos (tanto `const` o no) se pueden inicializar en sus definiciones (fuera de la declaración de clase). Por ejemplo:

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()
{
}

```

Acceso a elementos de matriz

Se puede obtener acceso a los elementos individuales de una matriz mediante el operador de subíndice de matriz (`[]`). Si usa el nombre de una matriz unidimensional sin un subíndice, se evalúa como un puntero al primer elemento de la matriz.

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.
}
```

Cuando se utilizan matrices multidimensionales, se pueden emplear varias combinaciones en las expresiones.

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.
}
```

En el código anterior, `multi` es una matriz tridimensional de tipo `double`. El puntero `p2multi` apunta a una matriz de tipo `double` de tamaño tres. En este ejemplo, la matriz se utiliza con uno, dos y tres subíndices. Aunque es más frecuente especificar todos los subíndices, como en la instrucción `cout`, a veces es útil seleccionar un subconjunto concreto de elementos de la matriz, como se muestra en las instrucciones que hay después de `cout`.

Sobrecarga del operador de subíndice

Al igual que otros operadores, el operador de subíndice (`[]`) lo puede volver a definir el usuario. El comportamiento predeterminado del operador de subíndice, si no está sobrecargado, consiste en combinar el nombre de la matriz y el subíndice usando el método siguiente:

```
*((array_name) + (subscript))
```

Como en toda suma que implica tipos de puntero, la escala se realiza automáticamente para que se produzca el ajuste correspondiente al tamaño del tipo. El valor resultante no es n bytes del origen de `array_name`; en su lugar, es el elemento n de la matriz. Para más información sobre esta conversión, consulte [Operadores de suma](#).

De igual forma, para las matrices multidimensionales, la dirección se deriva usando el método siguiente:

```
((array_name) + (subscript1 * max2 * max3 * ... * maxn) + (subscript2 * max3 * ... * maxn) + ... + subscriptn))
```

Matrices en expresiones

Cuando un identificador de un tipo de matriz aparece en una expresión distinta de `sizeof`, dirección de `(&)` o inicialización de una referencia, se convierte en un puntero al primer elemento de matriz. Por ejemplo:

C++

```
char szError1[] = "Error: Disk drive not ready.";
char *psz = szError1;
```

El puntero `psz` apunta al primer elemento de la matriz `szError1`. Las matrices, a diferencia de los punteros, no son valores L modificables. Por eso la siguiente asignación no es válida:

C++

```
szError1 = psz;
```

Consulte también

[std::array](#)

Referencias (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Una referencia (por ejemplo, un puntero) almacena la dirección de un objeto que se encuentra en otra parte de la memoria. A diferencia de un puntero, una referencia inicializada no puede hacer referencia a otro objeto ni establecerse en null. Hay dos tipos de referencias: referencias lvalue, que hacen referencia a una variable con nombre y referencias rvalue, que hacen referencia a un [objeto temporal](#). El operador & significa una referencia lvalue, mientras que el operador && significa una referencia rvalue o una referencia universal (lvalue o rvalue) según el contexto.

Las referencias se pueden declarar con la sintaxis siguiente:

```
[storage-class-specifiers] [cv-qualifiers] type-specifiers [ms-modifier] declarator  
[=expression];
```

Se puede usar cualquier declarador válido que especifique una referencia. A menos que la referencia sea una referencia a un tipo de función o matriz, se aplica la siguiente sintaxis simplificada:

```
[storage-class-specifiers] [cv-qualifiers] type-specifiers [& or &&] [cv-qualifiers]  
identifier [=expression];
```

Las referencias se declaran mediante la siguiente secuencia:

1. Los especificadores de la declaración:

- Un especificador de clase de almacenamiento opcional.
- Calificadores `const` y `volatile` opcionales.
- El especificador de tipo: el nombre de un tipo.

2. El declarador:

- Modificador opcional concreto de Microsoft. Para obtener más información, consulte [Modificadores específicos de Microsoft](#).
- El operador `&` o el operador `&&`.
- Calificadores `const` y/o `volatile` opcionales.
- El identificador.

3. Un inicializador opcional.

Las formas de declarador más complejas para punteros a matrices y funciones también se aplican a las referencias a matrices y funciones. Para más información, consulte [punteros](#).

Varios declaradores e inicializadores pueden aparecer en una lista separada por comas detrás de un único especificador de declaración. Por ejemplo:

C++

```
int &i;  
int &i, &j;
```

Las referencias, punteros y objetos se pueden declarar juntos:

C++

```
int &ref, *ptr, k;
```

Una referencia contiene la dirección de un objeto, pero se comporta sintácticamente como un objeto.

Observe que, en el siguiente programa, el nombre del objeto `s` y la referencia al objeto, `SRef`, se pueden usar de la misma forma:

Ejemplo

C++

```
// references.cpp  
#include <stdio.h>  
struct S {  
    short i;  
};  
  
int main() {  
    S s;      // Declare the object.  
    S& SRef = s;    // Declare 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  
4  
4
```

Consulte también

[Argumentos de función de tipo de referencia](#)

[Valores devueltos de función de tipo de referencia](#)

[Referencias a punteros](#)

Declarador de referencia a un valor L: &

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Contiene la dirección de un objeto, pero se comporta sintácticamente como un objeto.

Sintaxis

Lvalue-reference-type-id:

type-specifier-seq & attribute-specifier-seq opt *ptr-abstract-declarator* opt

Comentarios

Una referencia de valor L se puede considerar otro nombre para un objeto. Una declaración de referencia de valor L está formada por una lista opcional de especificadores seguida de un declarador de referencia. Una referencia debe inicializarse y no se puede cambiar.

Cualquier objeto cuya dirección se pueda convertir a un tipo de puntero especificado también se puede convertir a un tipo de referencia similar. Por ejemplo, cualquier objeto cuya dirección se pueda convertir al tipo `char *` también se puede convertir al tipo `char &`.

No confunda las declaraciones de referencia con el uso del [Operador address-of](#). Cuando `&identifier` va precedido de un tipo, como, `int` o `char`, `identifier` se declara como una referencia al tipo. Cuando `&identifier` no va precedido de un tipo, la utilización es la del operador address-of.

Ejemplo

En el ejemplo siguiente se muestra el declarador de referencia mediante la declaración de un objeto `Person` y una referencia a ese objeto. Dado que `rFriend` es una referencia a `myFriend`, actualizar una de las variables cambia el mismo objeto.

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
```

Consulte también

[Referencias](#)

[Argumentos de función de tipo de referencia](#)

[Valores devueltos de función de tipo de referencia](#)

[Referencias a punteros](#)

Declarador de referencia a un valor R:

&&

Artículo • 03/03/2023 • Tiempo de lectura: 13 minutos

Mantiene una referencia a una expresión de valor R.

Sintaxis

`rvalue-reference-type-id:`

`type-specifier-seq && attribute-specifier-seqoptptr-abstract-declaratoropt`

Comentarios

Las referencias de valor R permiten distinguir un valor L de un valor R. Las referencias de valor L y valor R son sintáctica y semánticamente similares, pero siguen reglas algo distintas. Para obtener más información sobre valores L y valores R, vea [Lvalues y Rvalues](#). Para obtener más información sobre las referencias lvalue, vea [Declarador de referencia lvalue: &](#).

Las secciones siguientes describen cómo las referencias de valor R admiten la implementación de *semántica de transferencia de recursos* y *reenvío directo*.

Semántica de transferencia de recursos

Las referencias de valor R admiten la implementación de la *semántica de transferencia de recursos*, que puede aumentar significativamente el rendimiento de las aplicaciones. La semántica de transferencia de recursos permite escribir código que transfiere recursos (tales como memoria asignada dinámicamente) de un objeto a otro. La semántica de transferencia de recursos funciona porque permite transferir recursos desde objetos temporales a los que no se puede hacer referencia en otra parte del programa.

Para implementar la semántica de transferencia de recursos, normalmente se proporciona un *constructor de movimiento* y opcionalmente, un operador de asignación de movimiento (`operator=`) a la clase. Las operaciones de copia y asignación cuyos orígenes son valores R aprovechan entonces automáticamente la semántica de transferencia de recursos. A diferencia del constructor de copia predeterminado, el compilador no proporciona un constructor de movimiento predeterminado. Para

obtener más información sobre cómo escribir y usar un constructor de movimiento, vea [Constructores de movimiento y operadores de asignación de movimiento](#).

También puede sobrecargar funciones y operadores normales para aprovechar la semántica de transferencia de recursos. Visual Studio 2010 presenta la semántica de movimiento a la biblioteca estándar de C++. Por ejemplo, la clase `string` implementa operaciones que usan semántica de transferencia de recursos. Considere el ejemplo siguiente que concatena varias cadenas e imprime el resultado:

C++

```
// 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;
}
```

Antes de Visual Studio 2010, cada llamada a `operator+` asigna y devuelve un nuevo objeto `string` temporal (un valor R). `operator+` no puede anexar una cadena a la otra porque no sabe si las cadenas de origen no son valores L o valores R. Si las cadenas de origen son valores l, es posible que se haga referencia a ellas en otro lugar del programa y, por tanto, no se deben modificar. Puede modificar `operator+` para tomar valores r mediante referencias rvalue, a las que no se puede hacer referencia en ningún otro lugar del programa. Con este cambio, `operator+` ahora puede anexar una cadena a otra. El cambio reduce significativamente el número de asignaciones de memoria dinámica que la clase `string` debe realizar. Para obtener más información sobre la `string` clase, vea [basic_string Clase](#).

La semántica de transferencia de recursos también ayuda cuando el compilador no puede utilizar la optimización del valor devuelto (RVO) o la optimización del valor devuelto con nombre (NRVO). En estos casos, el compilador llama al constructor de movimiento si el tipo lo define.

Para entender mejor la semántica de transferencia de recursos, considere el ejemplo de la inserción de un elemento en un objeto `vector`. Si se supera la capacidad del `vector` objeto, el `vector` objeto debe reasignar suficiente memoria para sus elementos y, a continuación, copiar cada elemento en otra ubicación de memoria para dejar espacio para el elemento insertado. Cuando una operación de inserción copia un elemento,

primero crea un nuevo elemento. A continuación, llama al constructor de copia para copiar los datos del elemento anterior al nuevo elemento. Por último, destruye el elemento anterior. La semántica de transferencia de recursos permite mover objetos directamente sin tener que realizar una asignación de gran consumo de memoria ni operaciones de copia.

Para aprovechar la semántica de transferencia de recursos en el ejemplo `vector`, puede escribir un constructor de movimiento para mover los datos de un objeto a otro.

Para obtener más información sobre la introducción de la semántica de movimiento a la biblioteca estándar de C++ en Visual Studio 2010, vea [Biblioteca estándar de C++](#).

Reenvío perfecto

El reenvío directo reduce la necesidad de funciones sobrecargadas y ayuda a evitar el problema de reenvío. El *problema de reenvío* puede ocurrir cuando se escribe una función genérica que acepta referencias como parámetros. Si pasa (o *reenvía*) estos parámetros a otra función, por ejemplo, si toma un parámetro de tipo `const T&`, la función llamada no puede modificar el valor de ese parámetro. Si la función genérica toma un parámetro de tipo `T&`, no se puede llamar a la función mediante un valor R (tal como un objeto temporal o un literal entero).

Normalmente, para solucionar este problema, debe proporcionar versiones sobrecargadas de la función genérica que acepten tanto `T&` como `const T&` para cada uno de sus parámetros. Como resultado, el número de funciones sobrecargadas aumenta exponencialmente con el número de parámetros. Las referencias de rvalue permiten escribir una versión de una función que acepta argumentos arbitrarios. Después, esa función puede reenviarlas a otra función como si se hubiera llamado directamente a la otra función.

Considere el ejemplo siguiente en el que se declaran cuatro tipos, `W`, `X`, `Y` y `Z`. El constructor de cada tipo toma una combinación distinta de referencias de lvalue `const` y no `const` como parámetros.

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&) {}
};
```

Supongamos que desea escribir una función genérica que genere objetos. En el siguiente ejemplo, se muestra una forma de escribir esta función:

C++

```
template <typename T, typename A1, typename A2>
T* factory(A1& a1, A2& a2)
{
    return new T(a1, a2);
}
```

En el ejemplo siguiente se muestra una llamada válida a la función `factory`.

C++

```
int a = 4, b = 5;
W* pw = factory<W>(a, b);
```

Sin embargo, el ejemplo siguiente no contiene una llamada válida a la función `factory`. Se debe a que `factory` toma referencias lvalue modificables como parámetros, pero se llama mediante rvalues:

C++

```
Z* pz = factory<Z>(2, 2);
```

Normalmente, para solucionar este problema se debe crear una versión sobrecargada de la función `factory` para cada combinación de los parámetros `A&` y `const A&`. Las referencias de valor R permiten escribir una versión de la función `factory`, como se muestra en el ejemplo siguiente:

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));
}
```

En este ejemplo se usan referencias de valor R como parámetros para la función `factory`. El propósito de la `std::forward` función es reenviar los parámetros de la función de fábrica al constructor de la clase de plantilla.

En el ejemplo siguiente se muestra la función `main` que usa la función `factory` revisada para crear instancias de las clases `W`, `X` y `Z`. La función `factory` revisada reenvía sus parámetros (ya sean valores L o valores R) al constructor de clase adecuado.

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;
}
```

Propiedades de referencias rvalue

Puede sobrecargar una función para que acepte una referencia de valor L y una referencia de valor R.

Mediante la sobrecarga de una función para que acepte una referencia de lvalue o una referencia de rvalue `const`, puede escribir código que distinga entre objetos no modificables (lvalue) y valores temporales modificables (rvalue). Puede pasar un objeto a una función que acepte una referencia de valor a menos que el objeto esté marcado como `const`. El ejemplo siguiente muestra la función `f`, que se sobrecarga para aceptar una referencia de valor L y una referencia de valor R. La función `main` llama a `f` con ambos valores L y un valor R.

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());
}

```

Este ejemplo produce el siguiente resultado:

Output

```

In f(const MemoryBlock&). This version can't modify the parameter.
In f(MemoryBlock&&). This version can modify the parameter.

```

En este ejemplo, la primera llamada a `f` pasa una variable local (un valor L) como argumento. La segunda llamada a `f` pasa un objeto temporal como argumento. Como no se puede hacer referencia al objeto temporal en otra parte del programa, la llamada se enlaza a la versión sobrecargada de `f` que acepta una referencia de rvalue, que es libre de modificar el objeto.

El compilador trata una referencia de valor R con nombre como un valor L y una referencia de valor R sin nombre como un valor R.

Las funciones que toman una referencia rvalue como parámetro tratan el parámetro como un valor l en el cuerpo de la función. El compilador trata una referencia con nombre rvalue como un valor lvalue. Se debe a que varias partes de un programa

pueden hacer referencia a un objeto con nombre. Es peligroso permitir que varias partes de un programa modifiquen o quiten recursos de ese objeto. Por ejemplo, si varias partes de un programa intentan transferir recursos del mismo objeto, solo la primera parte transferirá el recurso correctamente.

El ejemplo siguiente muestra la función `g`, que se sobre carga para aceptar una referencia de valor L y una referencia de valor R. La función `f` acepta una referencia de valor R como parámetro (una referencia de valor R con nombre) y devuelve una referencia de valor R (una referencia de valor R sin nombre). En la llamada a `g` desde `f`, la resolución de sobre carga selecciona la versión de `g` que acepta una referencia de valor L porque el cuerpo de `f` considera el parámetro como un valor L. En la llamada a `g` desde `main`, la resolución de sobre carga selecciona la versión de `g` que acepta una referencia de valor R porque `f` devuelve una referencia de valor R.

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()));
}
```

Este ejemplo produce el siguiente resultado:

C++

```
In g(const MemoryBlock&).
In g(MemoryBlock&&).
```

En este ejemplo, la función `main` pasa un rvalue a `f`. El cuerpo de `f` trata el parámetro con nombre como valor L. La llamada de `f` a `g` enlaza el parámetro a una referencia de valor L (la primera versión sobrecargada de `g`).

- Puede convertir un valor L en una referencia de valor R.

La función `std::move` de Biblioteca estándar de C++ permite convertir un objeto en una referencia de valor R a ese objeto. También puede usar la palabra clave `static_cast` para convertir un lvalue en una referencia de rvalue, como se muestra en el ejemplo siguiente:

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));
}
```

Este ejemplo produce el siguiente resultado:

C++

```
In g(const MemoryBlock&).
In g(MemoryBlock&&).
```

Las plantillas de función deducen sus tipos de argumento de plantilla y, a continuación, usan reglas de contracción de referencia.

Una plantilla de función que pase (o *reenvíe*) sus parámetros a otra función es un patrón común. Es importante saber cómo funciona la deducción de tipos de plantilla para plantillas de función que acepten referencias de rvalue.

Si el argumento de función es un valor R, el compilador deduce que el argumento será una referencia de valor R. Por ejemplo, supongamos que pasa una referencia rvalue a un objeto de tipo `x` a una plantilla de función que toma el tipo `T&&` como parámetro. La deducción del argumento de plantilla deduce `T` que es `x`, por lo que el parámetro tiene el tipo `x&&`. Si el argumento de función es un lvalue o un lvalue `const`, el compilador deduce que su tipo será una referencia de lvalue `const` o una referencia de lvalue de ese tipo.

En el ejemplo siguiente se declara una plantilla de estructura y, a continuación, se especializa para distintos tipos de referencia. La función `print_type_and_value` acepta una referencia de valor R como parámetro y lo reenvía a la versión especializada adecuada del método `S::print`. La función `main` muestra las diversas maneras de llamar al método `S::print`.

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());  
}
```

Este ejemplo produce el siguiente resultado:

C++

```
print<T&>: first  
print<const T&>: second  
print<T&&>: third  
print<const T&&>: fourth
```

Para resolver cada llamada a la función `print_type_and_value`, el compilador realiza primero la deducción de argumento de plantilla. El compilador aplica entonces las reglas de colapso de referencias cuando sustituye los tipos de parámetros por los argumentos deducidos de la plantilla. Por ejemplo, al pasar la variable local `s1` a la función `print_type_and_value` se provoca que el compilador genere la siguiente firma de función:

C++

```
print_type_and_value<string&>(string& && t)
```

El compilador usa reglas de contracción de referencias para reducir la firma:

C++

```
print_type_and_value<string&>(string& t)
```

Esta versión de la función `print_type_and_value` reenvía a continuación su parámetro a la versión especializada correcta del método `s::print`.

En la tabla siguiente se resumen las reglas de contracción de referencias para la deducción de tipos de argumento de plantilla:

| Tipo expandido | Tipo contraído |
|-------------------------------------|--------------------------|
| <code>T& &</code> | <code>T&</code> |
| <code>T& &&</code> | <code>T&</code> |
| <code>T&& &</code> | <code>T&</code> |
| <code>T&& &&</code> | <code>T&&</code> |

La deducción de argumento de plantilla es un elemento importante de la implementación del reenvío directo. En la sección Reenvío perfecto se describe con más detalle el [reenvío](#) perfecto.

Resumen

Las referencias de valor R distinguen los valores L de los valores R. Para mejorar el rendimiento de las aplicaciones, se puede eliminar la necesidad de asignaciones de memoria y operaciones de copia innecesarias. También permiten escribir una función que acepte argumentos arbitrarios. Esa función puede reenviarlas a otra función como si se hubiera llamado directamente a la otra función.

Consulte también

[Expresiones con operadores unarios](#)

[Declarador de referencia a un lvalue: &](#)

[Lvalues y Rvalues](#)

[Constructores de movimiento y operadores de asignación de movimiento \(C++\)](#)

[Biblioteca estándar de C++](#)

Argumentos de función de tipo de referencia

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Suele ser más eficaz pasar referencias, en lugar de objetos grandes, a las funciones. De este modo, el compilador puede pasar la dirección del objeto mientras mantiene la sintaxis que se habría utilizado para tener acceso al objeto. Considere el ejemplo siguiente, en el que se usa la estructura `Date`:

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;
}
```

El código anterior muestra que, para acceder a los miembros de una estructura pasada por referencia, se usa el operador de selección de miembro (.) en lugar del operador de selección de miembro de puntero (->).

Aunque los argumentos pasados como tipos de referencia se rigen por la sintaxis de los tipos que no son de puntero, mantienen una característica importante de los tipos de puntero: son modificables a menos que se declaren como `const`. Dado que el código anterior no tenía como intención modificar el objeto `date`, un prototipo de función más adecuado sería:

C++

```
long DateOfYear( const Date& date );
```

Este prototipo garantiza que la función `DateOfYear` no cambiará su argumento.

Cualquier función que se ajuste a un prototipo que toma un tipo de referencia puede aceptar un objeto del mismo tipo en su lugar, porque hay una conversión estándar de *typename* a *typename&*.

Consulte también

[Referencias](#)

Valores devueltos de función de tipo de referencia

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Las funciones se pueden declarar para que devuelvan un tipo de referencia. Hay dos motivos para realizar este tipo de declaración:

- La información que se va a devolver es un objeto tan grande que devolver una referencia es más eficaz que devolver una copia.
- El tipo de la función debe ser un valor L.
- El objeto al que se hace referencia no saldrá del ámbito cuando la función devuelva un valor.

Al igual que puede ser más eficaz pasar objetos grandes *a* funciones por referencia, también puede resultar más eficaz devolver objetos grandes *de* funciones por referencia. El protocolo de devolución de referencias elimina la necesidad de copiar el objeto en una ubicación temporal antes de que se devuelva.

Los tipos de valor devuelto de las referencias también pueden ser útiles cuando la función se debe evaluar como un valor L. La mayoría de los operadores sobrecargados pertenecen a esta categoría, especialmente el operador de asignación. Los operadores sobrecargados se explican en [Operadores sobrecargados](#).

Ejemplo

Considere el ejemplo `Point`:

```
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";
}

```

Output

Output

```

x = 7
y = 9

```

Observe que las funciones `x` e `y` se declaran de forma que devuelvan referencias. Estas funciones se pueden utilizar en cada lado de una instrucción de asignación.

Tenga en cuenta también que en `main`, el objeto `ThePoint` permanece en el ámbito y, por tanto, sus miembros de referencia continúan activos y se puede obtener acceso a ellos de forma segura.

Las declaraciones de tipos de referencia deben contener inicializadores excepto en los casos siguientes:

- Declaración `extern` explícita
- Declaración de un miembro de clase

- Declaración dentro de una clase
- Declaración de un argumento a una función o el tipo de valor devuelto de una función

Precaución sobre devolución de dirección

Si se declara un objeto en el ámbito local, ese objeto se destruirá cuando la función devuelva un valor. Si la función devuelve una referencia a ese objeto, esa referencia probablemente provocará una infracción de acceso en tiempo de ejecución si el llamador intenta usar la referencia nula.

C++

```
// C4172 means Don't do this!!!
Foo& GetFoo()
{
    Foo f;
    ...
    return f;
} // f is destroyed here
```

El compilador detecta una advertencia en este caso: `warning C4172: returning address of local variable or temporary`. Es posible que, en programas simples y en ocasiones, no se produzca ninguna infracción de acceso si el llamador tiene acceso a la referencia antes de que se sobrescriba la ubicación de memoria. Es simplemente una cuestión de suerte, así que haga caso a la advertencia.

Consulte también

[Referencias](#)

Referencias a punteros

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Las referencias a punteros se pueden declarar de la misma manera que las referencias a objetos. Una referencia a un puntero es un valor modificable que se usa como un puntero normal.

Ejemplo

En este ejemplo de código se muestra la diferencia entre usar un puntero a un puntero y una referencia a un puntero.

Las funciones `Add1` y `Add2` son funcionalmente equivalentes, aunque no se les llama de la misma manera. La diferencia es que `Add1` usa el direccionamiento indirecto doble, pero `Add2` utiliza la comodidad de una referencia a un puntero.

```
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( BTREE* 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( BTREE **Root, char *szToAdd ) {
    if ( (*Root) == 0 ) {
        (*Root) = new BTREE;
        (*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( BTREE*& Root, char *szToAdd ) {
    if ( Root == 0 ) {
        Root = new BTREE;
        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.

Consulte también

[Referencias](#)

Punteros (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Un puntero es una variable que almacena la dirección de memoria de un objeto. Los punteros se usan ampliamente en C y C++ para tres propósitos principales:

- para asignar nuevos objetos en el montón,
- para pasar funciones a otras funciones
- para iterar sobre elementos en matrices u otras estructuras de datos.

En la programación de estilo C, se usan *punteros básicos* para todos estos escenarios. Sin embargo, los punteros básicos son el origen de muchos errores de programación graves. Por lo tanto, se desaconseja encarecidamente su uso, excepto cuando proporcionan una ventaja significativa de rendimiento y no hay ambigüedad en cuanto a qué puntero es el *puntero propietario* que es responsable de eliminar el objeto. El lenguaje C++ moderno proporciona *punteros inteligentes* para asignar objetos, *iteradores* para recorrer estructuras de datos y *expresiones lambda* para pasar funciones. Al usar estas características del lenguaje y de la biblioteca en lugar de punteros básicos, hará que su programa sea más seguro, más fácil de depurar y más fácil de entender y mantener. Consulte [Punteros inteligentes, iteradores y Expresiones lambda](#) para obtener más información.

En esta sección

- [Punteros básicos](#)
- [Punteros const y volatile](#)
- [Operadores new y delete](#)
- [Punteros inteligentes](#)
- [Procedimiento Creación y uso de instancias unique_ptr](#)
- [Procedimiento Creación y uso de instancias shared_ptr](#)
- [Procedimiento Creación y uso de instancias weak_ptr](#)
- [Procedimiento Creación y uso de instancias CComPtr y CComQIPtr](#)

Consulte también

[Iteradores](#)

[Expresiones lambda](#)

Punteros básicos (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 9 minutos

Un *puntero* es un tipo de variable. Almacena la dirección de un objeto en memoria y se usa para acceder a ese objeto. Un *puntero básico* es un puntero cuya duración no está controlada por un objeto encapsulado, como un [puntero inteligente](#). Se puede asignar un puntero básico a la dirección de otra variable que no sea de puntero, o bien se le puede asignar un valor de [nullptr](#). Un puntero al que no se ha asignado un valor contiene datos aleatorios.

También se puede *desreferenciar* un puntero para recuperar el valor del objeto al que apunta. El *operador de acceso a miembros* proporciona acceso a los miembros de un objeto.

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
```

Un puntero puede apuntar a un objeto con tipo o a [void](#). Cuando un programa asigna un objeto en el [montón](#) en memoria, recibe la dirección de ese objeto en forma de puntero. Estos punteros se denominan *punteros propietarios*. Se debe usar un puntero propietario (o una copia de él) para liberar explícitamente el objeto asignado al montón cuando ya no sea necesario. Si no se libera la memoria, se produce una *fuga de memoria* y se representa esa ubicación de memoria como no disponible para cualquier otro programa de la máquina. La memoria asignada mediante [new](#) debe liberarse mediante [delete](#) (o [delete\[\]](#)). Para más información, consulte los [operadores new y delete](#).

C++

```
MyClass* mc = new MyClass(); // allocate object on the heap
mc->print(); // access class member
delete mc; // delete object (please don't forget!)
```

Un puntero (si no se declara como [const](#)) se puede incrementar o reducir para que apunte a otra ubicación de la memoria. Esta operación se denomina *aritmética de puntero*. Se usa en la programación de estilo C para iterar en elementos de matrices u otras estructuras de datos. No se puede hacer que un puntero [const](#) apunte a una

ubicación de memoria diferente y, en ese sentido, se parece a una [referencia](#). Para más información, consulte los artículos sobre los [punteros const y volatile](#).

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.
```

En sistemas operativos de 64 bits, un puntero tiene un tamaño de 64 bits. El tamaño del puntero de un sistema determina la cantidad de memoria direccionable que puede tener. Todas las copias de un puntero apuntan a la misma ubicación de memoria. Los punteros (junto con las referencias) se usan ampliamente en C++ para pasar objetos más grandes hacia las funciones y desde estas. A menudo es más eficaz copiar la dirección de un objeto que copiar todo el objeto. Al definir una función, especifique los parámetros de puntero como `const` a menos que quiera que la función modifique el objeto. En general, las referencias a `const` son la manera preferida de pasar objetos a funciones a menos que el valor del objeto pueda ser `nullptr`.

Los [punteros a funciones](#) permiten pasar funciones a otras funciones. Se usan para "devoluciones de llamada" en la programación de estilo C. La programación actual en C++ usa [expresiones lambda](#) con este fin.

Inicialización y acceso de miembros

En el ejemplo siguiente se muestra cómo declarar, inicializar y usar un puntero básico. Se inicializa mediante `new` para que apunte a un objeto asignado en el montón, el cual debe `delete` de manera explícita. En el ejemplo también se muestran algunos de los peligros asociados a punteros básicos. (Recuerde que este ejemplo es programación de estilo C y no C++ moderno).

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(pm);
pm->print(); // "Erika, 3"
pm2->print(); // "Erika, 3"

// Dereference the pointer and pass a copy
// of the pointed-to object to a function
func_B(*pm);
pm->print(); // "Erika, 3" (original not modified by function)

delete(pm); // don't forget to give memory back to operating system!
// delete(pm2); //crash! memory location was already deleted
}

```

Aritmética de punteros y matrices

Los punteros y matrices están estrechamente relacionados. Cuando una matriz se pasa por valor a una función, se pasa como puntero al primer elemento. En el ejemplo siguiente se muestran las siguientes propiedades importantes de punteros y matrices:

- El operador `sizeof` devuelve el tamaño total en bytes de una matriz.
- Para determinar el número de elementos, divida el total de bytes por el tamaño de un elemento.
- Cuando se pasa una matriz a una función, se *degrada* a un tipo de puntero.
- Cuando el operador `sizeof` se aplica a un puntero, este devuelve el tamaño del puntero, por ejemplo, 4 bytes en x86 o 8 bytes en x64.

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);
}

```

Ciertas operaciones aritméticas se pueden usar en punteros que no son `const` para que apunten a otra ubicación de memoria. Los punteros se incrementan y reducen mediante los operadores `++`, `+=`, `--` y `-=`. Esta técnica se puede usar en matrices y es especialmente útil en búferes de datos sin tipo. Un objeto `void*` se incrementa según el tamaño de un `char` (1 byte). Un puntero con tipo se incrementa según el tamaño del tipo al que apunta.

En el ejemplo siguiente se muestra cómo se puede usar la aritmética de punteros para acceder a píxeles individuales en un mapa de bits en Windows. Anote el uso de `new` y `delete`, y el operador de desreferencia.

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();
}

```

Punteros `void*`

Un puntero a `void` simplemente apunta a una ubicación de memoria sin procesar. A veces es necesario usar punteros `void*`, por ejemplo, al pasar entre el código de C++ y las funciones de C.

Cuando se convierte un puntero con tipo en un puntero `void`, el contenido de la ubicación de memoria no cambia. Sin embargo, se pierde la información de tipo, por lo que no se pueden realizar operaciones de incremento ni de reducción. Una ubicación de memoria se puede convertir, por ejemplo, de `MyClass*` a `void*` y de nuevo a `MyClass*`. Estas operaciones son intrínsecamente propensas a errores y requieren mucho cuidado con los errores void. La programación en C++ moderna desaconseja el uso de punteros `void` en casi todas las circunstancias.

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);
}

```

Punteros a funciones

En la programación de estilo C, los punteros de función se usan principalmente para pasar funciones a otras funciones. Esta técnica permite al autor de la llamada personalizar el comportamiento de una función sin modificarla. En C++ actual, las

expresiones lambda proporcionan la misma funcionalidad con mayor seguridad de tipos y otras ventajas.

Una declaración de puntero de función especifica la firma que debe tener la función a la que apunta:

```
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);
```

En el ejemplo siguiente se muestra una función `combine` que toma como parámetro cualquier función que acepte un `std::string` y devuelva un `std::string`. Según la función que se pase a `combine`, se antepone o se anexa una cadena.

```
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";
}
```

Consulte también

[Punteros inteligentes](#)[Operador de direccionamiento indirecto: *](#)

[Operador Address-of: &](#)

[Aquí está otra vez C++](#)

punteros const y volatile

Artículo • 26/09/2022 • Tiempo de lectura: 3 minutos

Las palabras clave de `const` y `volatile` cambian el modo en que se tratan los punteros. La palabra clave `const` especifica que el puntero no se puede modificar después de la inicialización; desde ese momento, el puntero se protege de cualquier modificación.

La palabra clave `volatile` especifica que el valor asociado al nombre que va a continuación se puede modificar con acciones que no sean las de la aplicación del usuario. Por consiguiente, la palabra clave `volatile` es útil para declarar objetos en memoria compartida a los que puedan obtener acceso varios procesos o áreas de datos globales utilizados para la comunicación con rutinas de servicio de interrupción.

Si un nombre se declara como `volatile`, el compilador recarga el valor de la memoria cada vez que el programa tiene acceso al mismo. Esto reduce considerablemente las posibles optimizaciones. Sin embargo, cuando el estado de un objeto puede cambiar de forma inesperada, es la única forma garantizar un rendimiento predecible del programa.

Para declarar el objeto al que señala el puntero como `const` o `volatile`, utilice una declaración con el formato:

C++

```
const char *cpch;
volatile char *vpch;
```

Para declarar el valor del puntero (es decir, la dirección real almacenada en el puntero) como `const` o `volatile`, utilice una declaración con el formato:

C++

```
char * const pchc;
char * volatile pchv;
```

El lenguaje C++ evita asignaciones que pudieran permitir la modificación de un objeto o un puntero declarado como `const`. Estas asignaciones quitarían la información con la que se declaró el objeto o puntero, infringiendo así la intención de la declaración original. Considere las siguientes declaraciones:

C++

```
const char cch = 'A';
char ch = 'B';
```

Dadas las declaraciones anteriores de dos objetos (`cch`, de tipo `const char` y `ch`, de tipo `char`), la declaración o las inicializaciones siguientes son válidas:

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;
```

La declaración o inicializaciones siguientes son erróneas.

C++

```
char *pch2 = &cch;    // Error
char *const pch3 = &cch;    // Error
```

La declaración de `pch2` declara un puntero a través del cual podría modificarse un objeto constante y, por consiguiente, no se permite. La declaración de `pch3` especifica que el puntero es constante, no el objeto; la declaración no se permite por la misma razón que la declaración de `pch2`.

Las ocho asignaciones siguientes muestran la asignación a través de un puntero y cómo se cambia el valor del puntero para las declaraciones anteriores; por ahora, supongamos que la inicialización era correcta para las declaraciones de `pch1` a `pch8`.

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
```

Los punteros declarados como `volatile`, o como una combinación de `const` y `volatile`, siguen las mismas reglas.

Los punteros a objetos `const` suelen utilizarse en declaraciones de función como esta:

C++

```
errno_t strcpy_s( char *strDestination, size_t numberofElements, const char *strSource );
```

La instrucción anterior declara una función, `strcpy_s`, donde dos de los tres argumentos son de tipo puntero a `char`. Dado que los argumentos se pasan por referencia y no por valor, la función podría modificar libremente `strDestination` y `strSource` si `strSource` no se declaró como `const`. La declaración de `strSource` como `const` garantiza al llamador que la función llamada no puede cambiar `strSource`.

ⓘ Nota

Dado que hay una conversión estándar de `typename*` a `const typename*`, se permite pasar un argumento de tipo `char *` a `strcpy_s`. Sin embargo, no sucede lo mismo a la inversa; no existe ninguna conversión implícita para quitar el atributo `const` de un objeto o puntero.

Un puntero `const` de un tipo determinado se puede asignar a un puntero del mismo tipo. Sin embargo, un puntero que no es `const` no se puede asignar a un puntero `const`. El código siguiente muestra las asignaciones correctas e incorrectas:

C++

```
// const_pointer.cpp
int *const cpObject = 0;
int *pObject;

int main() {
    pObject = cpObject;
    cpObject = pObject; // C3892
}
```

En el ejemplo siguiente se muestra cómo declarar un objeto como `const` si tiene un puntero a un puntero a un objeto.

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;
}
```

Consulte también

[PunterosPunteros básicos](#)

Operadores `new` y `delete`

Artículo • 03/03/2023 • Tiempo de lectura: 8 minutos

C++ admite la asignación dinámica y la desasignación de objetos mediante los operadores `new` y `delete`. Estos operadores asignan memoria para los objetos de un grupo denominado *almacén libre* (también conocido como *montón*). El operador `new` llama a la función especial `operator new` y el operador `delete` llama a la función especial `operator delete`.

Para obtener una lista de los archivos de biblioteca que componen la biblioteca en tiempo de ejecución de C y la biblioteca estándar de C++, consulte [Características de la biblioteca CTR](#).

El operador `new`

El compilador traduce una instrucción como esta en una llamada a la función `operator new`:

```
C++  
char *pch = new char[BUFFER_SIZE];
```

Si la solicitud es para cero bytes de almacenamiento, `operator new` devuelve un puntero a un objeto distinto. Es decir, al llamar repetidamente a `operator new` se devuelven punteros diferentes.

Si no hay memoria suficiente para la solicitud de asignación, `operator new` produce una excepción `std::bad_alloc`. O bien, devuelve `nullptr` si ha usado el formulario de colocación de `new(std::nothrow)` o si ha vinculado en compatibilidad con `operator new` sin inicio. Para obtener más información, consulte [Comportamiento de los errores de asignación](#).

En la tabla siguiente se describen los dos ámbitos de las funciones `operator new`.

Ámbito de las funciones `operator new`

| Operador | Ámbito |
|-----------------------------|--------|
| <code>::operator new</code> | Global |

| Operador | Ámbito |
|--|--------|
| <code>class-name ::operator new</code> | Clase |

El primer argumento de `operator new` debe ser de tipo `size_t`, y el tipo de valor devuelto siempre será `void*`.

Se llama a la función global `operator new` cuando el operador `new` se usa para asignar objetos de tipos integrados, objetos de tipo de clase que no contienen funciones `operator new` definidas por el usuario y matrices de cualquier tipo. Cuando el operador `new` se usa para asignar objetos de un tipo de clase en la que se ha definido `operator new`, se llama a la función `operator new` de esa clase.

Una función `operator new` definida para una clase es una función miembro estática (que no puede ser virtual) que oculta la función global `operator new` para los objetos de ese tipo de clase. Imagine que se usa `new` para asignar y establecer la memoria en un valor determinado:

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;
}
```

El argumento proporcionado entre paréntesis a `new` se pasa a `Blanks::operator new` como el argumento `chInit`. Sin embargo, se oculta la función global `operator new`,

haciendo que código como el siguiente genere un error:

C++

```
Blanks *SomeBlanks = new Blanks;
```

El compilador admite los operadores `new` y `delete` de la matriz de miembro en una declaración de clase. Por ejemplo:

C++

```
class MyClass
{
public:
    void * operator new[] (size_t)
    {
        return 0;
    }
    void operator delete[] (void*)
    {
    }
};

int main()
{
    MyClass *pMyClass = new MyClass[5];
    delete [] pMyClass;
}
```

Comportamiento de los errores de asignación

La función `new` de la biblioteca estándar de C++ admite el comportamiento especificado en el estándar de C++ desde C++98. Cuando no hay memoria suficiente para una solicitud de asignación, `operator new` produce una excepción `std::bad_alloc`.

El código de C++ anterior devolvió un puntero nulo para una asignación con errores. Si tiene código que espera la versión no iniciada de `new`, vincule el programa con `nothrownew.obj`. El archivo `nothrownew.obj` reemplaza al `operator new` global con una versión que devuelve `nullptr` si se produce un error en una asignación. `operator new` ya no produce `std::bad_alloc`. Para obtener más información sobre `nothrownew.obj` y otros archivos de opciones del enlazador, consulte [Opciones de vínculo](#).

No se puede mezclar código que comprueba si hay excepciones del `operator new` global con código que comprueba si hay punteros nulos en la misma aplicación. Sin embargo, sigue pudiendo crear un elemento `operator new` local de clase que se

comporte de forma diferente. Esta posibilidad significa que el compilador debe actuar defensivamente de manera predeterminada e incluir comprobaciones para devoluciones de punteros nulos en las llamadas a `new`. Para obtener más información sobre cómo optimizar estas comprobaciones del compilador, consulte [/Zc:throwingnew](#).

Controlar la memoria insuficiente

La forma en que se prueba una asignación con errores de una expresión `new` depende de si se usa el mecanismo de excepción estándar o se usa una devolución `nullptr`. C++ estándar espera que un asignador produzca `std::bad_alloc` o una clase derivada de `std::bad_alloc`. Puede controlar una excepción como esta, como se muestra en este ejemplo:

C++

```
#include <iostream>
#include <new>
using namespace std;
#define BIG_NUMBER 1000000000LL
int main() {
    try {
        int *pI = new int[BIG_NUMBER];
    }
    catch (bad_alloc& ex) {
        cout << "Caught bad_alloc: " << ex.what() << endl;
        return -1;
    }
}
```

Cuando se usa la forma `nothrow` de `new`, puede probar si se produce un error de asignación, como se muestra en este ejemplo:

C++

```
#include <iostream>
#include <new>
using namespace std;
#define BIG_NUMBER 1000000000LL
int main() {
    int *pI = new(nothrow) int[BIG_NUMBER];
    if ( pI == nullptr ) {
        cout << "Insufficient memory" << endl;
        return -1;
    }
}
```

Puede probar si se produce una asignación de memoria con errores cuando haya usado el archivo `nothrownew.obj` para reemplazar el `operator new` global, como se muestra aquí:

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;
    }
}
```

Puede proporcionar un controlador para las solicitudes de asignación de memoria con errores. Es posible escribir una rutina de recuperación personalizada para controlar este tipo de errores. Por ejemplo, podría liberar memoria reservada y, a continuación, permitir que la asignación se ejecute de nuevo. Para más información, consulte [_set_new_handler](#).

El operador `delete`

La memoria que se asigna dinámicamente mediante el operador `new` se puede liberar con el operador `delete`. El operador `delete` llama a la función `operator delete`, que libera memoria para que la use el grupo disponible. El uso del operador `delete` también ocasiona una llamada al destructor de clase (si existe alguno).

Hay funciones `operator delete` globales y de ámbito de clase. Solo se puede definir una función `operator delete` para una clase dada; si se define, oculta la función `operator delete` global. La función `operator delete` global siempre se llama para las matrices de cualquier tipo.

La función `operator delete` global. Existen dos formularios para las funciones `operator delete` global y `operator delete` de miembro de clase:

C++

```
void operator delete( void * );
void operator delete( void *, size_t );
```

Solo uno de los dos formularios anteriores puede estar presente para una clase determinada. El primer formulario toma un único argumento del tipo `void *`, que contiene un puntero al objeto que se debe desasignar. El segundo formulario, una desasignación con tamaño, toma dos argumentos: el primero es un puntero al bloque de memoria que se va a desasignar y el segundo es el número de bytes que se van a desasignar. El tipo de valor devuelto de ambos formularios es `void` (`operator delete` no puede devolver un valor).

La intención del segundo formulario es acelerar la búsqueda de la categoría de tamaño correcta del objeto que se va a eliminar. Esta información a menudo no se almacena cerca de la propia asignación y es probable que no se almacene en caché. El segundo formulario resulta especialmente útil cuando se usa una función `operator delete` de una clase base para eliminar un objeto de una clase derivada.

La función `operator delete` es estática, por lo que no puede ser virtual. La función `operator delete` obedece al control de acceso, tal como se describe en [Control de acceso a miembros](#).

En el ejemplo siguiente se muestran las funciones `operator new` y `operator delete` definidas por el usuario, diseñadas para registrar asignaciones y desasignaciones de memoria:

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;
}

```

El código anterior se puede utilizar para detectar "fugas de memoria", es decir, la memoria que se asigna en el almacén libre pero que nunca se libera. Para detectar fugas, los operadores `new` y `delete` globales se redefinen para determinar la asignación y desasignación de memoria.

El compilador admite los operadores `new` y `delete` de la matriz de miembro en una declaración de clase. Por ejemplo:

C++

```

// 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;
}

```

Punteros inteligentes (C++ moderno)

Artículo • 26/09/2022 • Tiempo de lectura: 8 minutos

En la programación del lenguaje C++ actual, la biblioteca estándar incluye *punteros inteligentes*, que se utilizan para asegurarse de que los programas están libres de memoria y de pérdidas de recursos y son seguros ante excepciones.

Usos de los punteros inteligentes

Los punteros inteligentes se definen en el espacio de nombres `std` del archivo de encabezado `<memory>`. Son cruciales en la expresión de programación **RAII** o *Resource Acquisition Is Initialization*. El objetivo principal de esta expresión es asegurarse de que la adquisición de recursos ocurre al mismo tiempo que se inicializa el objeto, de manera que todos los recursos del objeto se creen y se dispongan en una sola línea de código. En la práctica, el principio básico RAII consiste en proporcionar la propiedad de cualquier recurso asignado por montón (por ejemplo, memoria asignada dinámicamente o identificadores de objetos del sistema) a un objeto asignado a la pila cuyo destructor contiene código para eliminar o liberar el recurso, además de cualquier código asociado de limpieza.

En la mayoría de los casos, cuando se inicializa un puntero o un identificador de recursos sin formato para apuntar a un recurso real, el puntero se pasa inmediatamente a un puntero inteligente. En el lenguaje C++ actual, los punteros sin formato se utilizan únicamente en pequeños bloques de código de ámbito limitado, bucles o funciones del asistente donde el rendimiento es crucial y no hay ninguna posibilidad de confusión sobre la propiedad.

En el ejemplo siguiente se compara una declaración de puntero sin formato con una declaración de puntero inteligente.

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.
```

Como se muestra en el ejemplo, un puntero inteligente es una plantilla de clase que se declara en la pila y se inicializa con un puntero sin formato que apunta a un objeto asignado por montón. Una vez que se inicializa el puntero inteligente, se convierte en propietario del puntero sin formato. Esto significa que el puntero inteligente es responsable de eliminar la memoria que el puntero sin formato especifica. El destructor del puntero inteligente contiene la llamada de eliminación y, dado que el puntero inteligente se declara en la pila, su destructor se invoca cuando el puntero inteligente sale del ámbito, incluso si se produce una excepción en alguna parte que se encuentre más arriba en la pila.

Para acceder al puntero encapsulado, utilice los conocidos operadores de puntero `->` y `*`, que la clase del puntero inteligente sobrecarga para devolver el puntero sin formato encapsulado.

La expresión del puntero inteligente de C++ se asemeja a la creación de objetos en lenguajes como C#: se crea el objeto y después se permite al sistema que se ocupe de eliminarlo en el momento correcto. La diferencia es que ningún recolector de elementos no utilizados independiente se ejecuta en segundo plano; la memoria se administra con las reglas estándar de ámbito de C++, de modo que el entorno en tiempo de ejecución es más rápido y eficaz.

ⓘ Importante

Cree siempre punteros inteligentes en una línea de código independiente, nunca en una lista de parámetros, para que no se produzca una pérdida de recursos imperceptible debido a algunas reglas de asignación de la lista de parámetros.

En el ejemplo siguiente se muestra cómo se puede utilizar un tipo de puntero inteligente `unique_ptr` de la biblioteca estándar de C++ para encapsular un puntero a un objeto grande.

```

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.

```

En el ejemplo se muestran los pasos básicos siguientes para utilizar punteros inteligentes.

1. Declare el puntero inteligente como variable automática (local). (No utilice la expresión `new` o `malloc` en el propio puntero inteligente).
2. En el parámetro de tipo, especifique el tipo al que apunta el puntero encapsulado.
3. Pase un puntero sin formato al objeto `new`-ed en el constructor de puntero inteligente. (Algunas funciones de utilidad o constructores de puntero inteligente hacen esto por usted.)
4. Utilice los operadores sobrecargados `->` y `*` para tener acceso al objeto.
5. Deje que el puntero inteligente elimine el objeto.

Los punteros inteligentes están diseñados para ser lo más eficaces posible tanto en términos de memoria como de rendimiento. Por ejemplo, el único miembro de datos de `unique_ptr` es el puntero encapsulado. Esto significa que `unique_ptr` tiene exactamente el mismo tamaño que ese puntero, cuatro u ocho bytes. El acceso al puntero encapsulado a través del puntero inteligente sobrecargado `*` y los operadores `>` no es mucho más lento que el acceso directo a los punteros sin formato.

Los punteros inteligentes tienen sus propias funciones miembro, a las que se accede mediante la notación "punto". Por ejemplo, algunos punteros inteligentes de la biblioteca estándar de C++ tienen una función miembro de restablecimiento que libera

la propiedad del puntero. Esto es útil cuando se quiere liberar la memoria que es propiedad del puntero inteligente antes de que el puntero inteligente salga del ámbito, tal y como se muestra en el ejemplo siguiente.

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...
}
```

Los punteros inteligentes suelen proporcionar un mecanismo para acceder directamente al puntero sin formato. Los punteros inteligentes de la biblioteca estándar de C++ tienen una función miembro `get` con este propósito y `CComPtr` tiene un miembro de clase `p` público. Si proporciona acceso directo al puntero subyacente, puede utilizar el puntero inteligente para administrar la memoria en el propio código y continuar pasando el puntero sin formato en un código que no admite punteros inteligentes.

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());
}
```

Clases de punteros inteligentes

En la sección siguiente se resumen los distintos tipos de punteros inteligentes disponibles en el entorno de programación de Windows y se describe cuándo utilizarlos.

Punteros inteligentes de la biblioteca estándar de C++

Utilice estos punteros inteligentes como primera opción para encapsular punteros a los objetos estándar de C++ (POCO).

- `unique_ptr`

Permite exactamente un único propietario del puntero subyacente. Utilice esta opción como predeterminada para los objetos POCO, a menos que sepa con certeza que necesita un objeto `shared_ptr`. Puede moverse a un nuevo propietario, pero no se puede copiar ni compartir. Sustituye a `auto_ptr`, que está desusado. Comparado con `boost::scoped_ptr`, `unique_ptr` es pequeño y eficiente; el tamaño es un puntero y admite referencias `rvalue` para la inserción y recuperación rápidas de colecciones de bibliotecas estándar de C++. Archivo de encabezado: `<memory>`. Para obtener más información, vea [Cómo: Crear y usar instancias de `ptr` Instances](#) y [unique_ptr Class](#).

- `shared_ptr`

Puntero inteligente con recuento de referencias. Utilícelo cuando desee asignar un puntero sin formato a varios propietarios, por ejemplo, cuando devuelve una copia de un puntero desde un contenedor pero desea conservar el original. El puntero sin formato no se elimina hasta que todos los propietarios de `shared_ptr` han salido del ámbito o, de lo contrario, han renunciado a la propiedad. El tamaño es dos punteros: uno para el objeto y otro para el bloque de control compartido que contiene el recuento de referencias. Archivo de encabezado: `<memory>`. Para obtener más información, vea [Cómo: Crear y usar instancias de `shared_ptr` y clase `shared_ptr`](#).

- `weak_ptr`

Puntero inteligente de caso especial para usarlo junto con `shared_ptr`. `weak_ptr` proporciona acceso a un objeto que pertenece a una o varias instancias de `shared_ptr`, pero no participa en el recuento de referencias. Utilícelo cuando desee observar un objeto, pero no quiere que permanezca activo. Es necesario en algunos casos para interrumpir las referencias circulares entre instancias de `shared_ptr`. Archivo de encabezado: `<memory>`. Para obtener más información, vea [Cómo: Crear y usar instancias de `weak_ptr` y clase `weak_ptr`](#).

Punteros inteligentes para objetos COM (programación clásica de Windows)

Cuando trabaje con objetos COM, encapsule los punteros de interfaz en un tipo de puntero inteligente adecuado. Active Template Library (ATL) define varios punteros inteligentes para propósitos diferentes. También puede usar el tipo de puntero inteligente `_com_ptr_t`, que el compilador utiliza cuando crea clases contenedoras de archivos .tlb. Es la mejor opción si no desea incluir los archivos de encabezado ATL.

[CComPtr \(clase\)](#)

Utilice esta opción a menos que no puede usar ATL. Realiza el recuento de referencias mediante los métodos `AddRef` y de `Release`. Para obtener más información, consulte [Cómo: Crear y usar instancias de CComPtr y CComQIPtr](#).

[CComQIPtr \(clase\)](#)

Se parece a `CComPtr`, pero también proporciona la sintaxis simplificada para llamar a `QueryInterface` en objetos COM. Para obtener más información, consulte [Cómo: Crear y usar instancias de CComPtr y CComQIPtr](#).

[CComHeapPtr \(clase\)](#)

Puntero inteligente a objetos que utilizan `CoTaskMemFree` para liberar memoria.

[CComGITPtr \(clase\)](#)

Puntero inteligente para las interfaces que se obtienen de la tabla de interfaz global (GIT).

[_com_ptr_t \(Clase\)](#)

Se parece a `CComQIPtr` en funcionalidad, pero no depende de los encabezados ATL.

Punteros inteligentes ATL para objetos POCO

Además de punteros inteligentes para los objetos COM, ATL también define punteros inteligentes y colecciones de punteros inteligentes para objetos estándar de C++ (POCO). En la programación clásica de Windows, estos tipos son alternativas útiles a las colecciones de bibliotecas estándar de C++, especialmente cuando no se requiere portabilidad de código o cuando no se quieren mezclar los modelos de programación de la biblioteca estándar de C++ y ATL.

[CAutoPtr \(clase\)](#)

Puntero inteligente que exige una propiedad única al transferir la propiedad en la copia. Puede compararse con la clase `std::auto_ptr` en desuso.

[CHheapPtr \(clase\)](#)

Puntero inteligente para objetos asignados mediante la función de C `malloc`.

[CAutoVectorPtr \(clase\)](#)

Puntero inteligente para matrices que se asignan mediante `new[]`.

[CAutoPtrArray \(clase\)](#)

Clase que encapsula una matriz de elementos `CAutoPtr`.

[CAutoPtrList \(clase\)](#)

Clase que encapsula los métodos para manipular una lista de nodos de `CAutoPtr`.

Consulte también

[Punteros](#)

[Referencia del lenguaje C++](#)

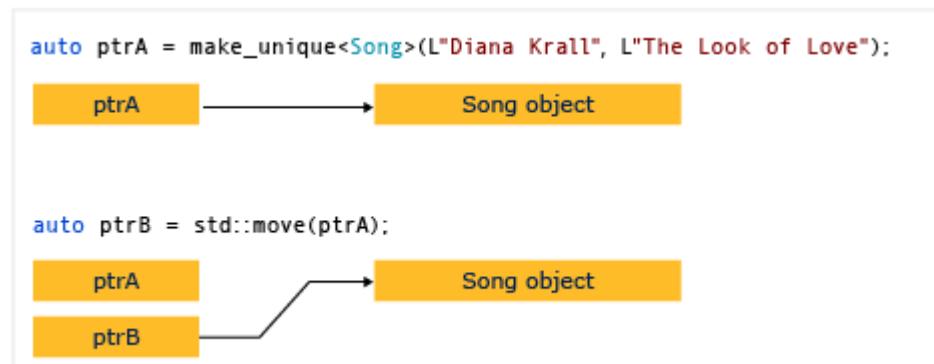
[Biblioteca estándar de C++](#)

Procedimiento Creación y uso de instancias unique_ptr

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Un `unique_ptr` no comparte el puntero. No se puede copiar en otro `unique_ptr`, pasar por valor a una función ni utilizar en ningún algoritmo de la Biblioteca estándar de C++ que requiera hacer copias. Un `unique_ptr` solo se puede mover. Esto significa que la propiedad del recurso de memoria se transfiere a otro `unique_ptr` y el `unique_ptr` original deja de poseerlo. Se recomienda limitar un objeto a un propietario, porque la propiedad múltiple agrega complejidad a la lógica del programa. Por consiguiente, si necesita un puntero inteligente para un objeto de C++ sin formato, utilice `unique_ptr`, y cuando construya un `unique_ptr`, utilice la función del asistente `make_unique`.

El diagrama siguiente muestra la transferencia de propiedad entre dos instancias de `unique_ptr`.



`unique_ptr` se define en el encabezado `<memory>` de la Biblioteca estándar de C++. Es exactamente tan eficaz como un puntero sin procesar y se puede usar en contenedores de la Biblioteca estándar de C++. La adición de las instancias de `unique_ptr` a los contenedores STL es eficaz porque el constructor de movimiento de `unique_ptr` elimina la necesidad de una operación de copia.

Ejemplo 1

En el ejemplo siguiente se muestra cómo crear instancias de `unique_ptr` y pasárselas entre funciones.

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");
}

```

Estos ejemplos muestran esta característica básica de `unique_ptr`: se puede mover, pero no copiar. "Mover" transfiere la propiedad a un nuevo `unique_ptr` y restablece el antiguo `unique_ptr`.

Ejemplo 2

En el ejemplo siguiente se muestra cómo crear instancias del objeto `unique_ptr` y usarlas en un vector.

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;
    }
}

```

En el intervalo del bucle, observe que `unique_ptr` se pasa por referencia. Si intenta pasar el parámetro por valor aquí, el compilador producirá un error porque se elimina el constructor de copias `unique_ptr`.

Ejemplo 3

En el siguiente ejemplo se muestra cómo se inicializa un `unique_ptr` que es miembro de una clase.

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();
    }
};
```

Ejemplo 4

Puede utilizar `make_unique` para crear `unique_ptr` a una matriz, pero no puede utilizar `make_unique` para inicializar los elementos de matriz.

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;
}
```

Para obtener más ejemplos, consulte [make_unique](#).

Consulte también

[Punteros inteligentes \(C++ moderno\)](#)

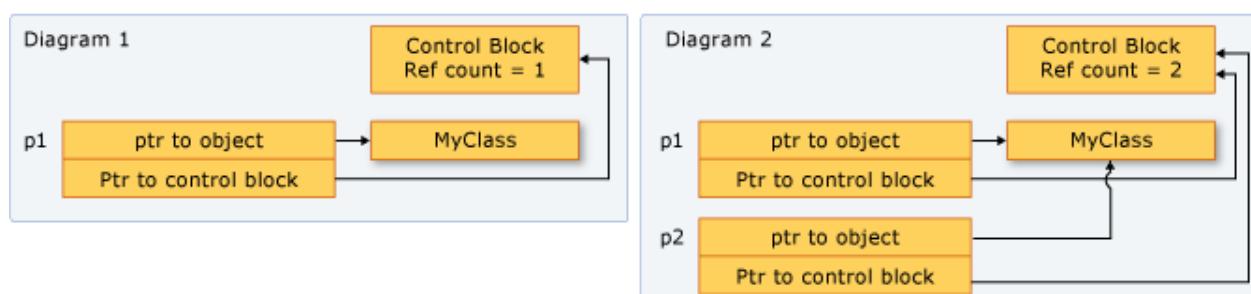
[make_unique](#)

Cómo: crear y utilizar instancias de shared_ptr

Artículo • 03/03/2023 • Tiempo de lectura: 7 minutos

El tipo `shared_ptr` es puntero inteligente de la biblioteca estándar de C++ que está diseñado para escenarios en los que más de un propietario tendrá que administrar la duración del objeto en memoria. Después de inicializar `shared_ptr`, puede copiarlo, pasarlo por valor en argumentos de función y asignarlo a otras instancias de `shared_ptr`. Todas las instancias apuntan al mismo objeto y el acceso compartido a un "bloque de control" aumenta o disminuye el recuento de referencias siempre que un nuevo `shared_ptr` se agrega, se sale del ámbito o se restablece. Cuando el recuento de referencias llega a cero, el bloque de control elimina el recurso de memoria y se elimina a sí mismo.

En la ilustración siguiente se muestran varias instancias de `shared_ptr` que apuntan a una ubicación de memoria.



Configuración de muestra

En los ejemplos siguientes se da por hecho que ha incluido los encabezados requeridos y declarado los tipos necesarios, tal como se muestra aquí:

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
}

```

Ejemplo 1

Siempre que sea posible, utilice la función `make_shared` para crear `shared_ptr` cuando se cree el recurso de memoria por primera vez. `make_shared` es seguro para excepciones. Utiliza la misma llamada para asignar memoria para el bloque de control y el recurso, lo que reduce la sobrecarga de la construcción. Si no utiliza `make_shared`, debe usar una expresión `new` explícita para crear el objeto antes de pasarlo al constructor de `shared_ptr`. En el ejemplo siguiente se muestran varias maneras de declarar e inicializar `shared_ptr` junto con un nuevo objeto.

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");
```

Ejemplo 2

En el ejemplo siguiente se muestra cómo declarar e inicializar las instancias de `shared_ptr` que adquieren la propiedad compartida de un objeto que otro `shared_ptr` ya ha asignado. Suponga que `sp2` es un puntero `shared_ptr` inicializado.

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);
```

Ejemplo 3

`shared_ptr` también es útil en los contenedores de la biblioteca estándar de C++ cuando se utilizan algoritmos que copian elementos. Puede ajustar los elementos en `shared_ptr` y copiarlos en otros contenedores, pero debe tener en cuenta que la memoria subyacente es válida mientras se necesita, y no más. En el ejemplo siguiente se muestra cómo usar el algoritmo `remove_copy_if` en las instancias de `shared_ptr` en un vector.

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;
}
```

Ejemplo 4

Puede utilizar `dynamic_pointer_cast`, `static_pointer_cast` y `const_pointer_cast` para convertir `shared_ptr`. Estas funciones se parecen a los operadores `dynamic_cast`, `static_cast` y `const_cast`. En el ejemplo siguiente se muestra cómo probar el tipo derivado de cada elemento en un vector de `shared_ptr` de clases base y, a continuación, copiar los elementos y mostrar información sobre ellos.

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;  
}
```

Ejemplo 5

Puede pasar `shared_ptr` a otra función de las maneras siguientes:

- Pasar `shared_ptr` por valor. Esto invoca el constructor de copias, incrementa el recuento de referencias y convierte al destinatario en propietario. Hay una pequeña cantidad de sobrecarga en esta operación, que puede ser significativa en función del número de objetos `shared_ptr` que se pasen. Utilice esta opción cuando el contrato de código implícito o explícito entre el llamador y el destinatario requiera que el destinatario sea propietario.
- Pasar `shared_ptr` por referencia o referencia const. En este caso, el recuento de referencias no se incrementa y el destinatario puede tener acceso al puntero siempre que el llamador no salga del ámbito. O bien, el destinatario puede decidir crear un puntero `shared_ptr` basado en la referencia y pasar a ser el propietario compartido. Utilice esta opción cuando el llamador no tiene conocimiento del destinatario o cuando se debe pasar `shared_ptr` y desea evitar la operación de copia por razones de rendimiento.
- Pasar el puntero subyacente o una referencia al objeto subyacente. Esto permite al destinatario utilizar el objeto, pero no permite compartir la propiedad ni extender la duración. Si el destinatario crea un puntero `shared_ptr` a partir del puntero sin formato, el nuevo elemento `shared_ptr` será independiente del original y no controlará el recurso subyacente. Utilice esta opción cuando el contrato entre el llamador y el destinatario especifique claramente que el llamador conserva la propiedad de la duración de `shared_ptr`.
- Cuando decida cómo pasar un puntero `shared_ptr`, determine si el destinatario tiene que compartir la propiedad del recurso subyacente. Un "propietario" es un objeto o función que puede mantener el recurso subyacente activo mientras lo necesite. Si el llamador tiene que garantizar que el destinatario pueda extender la vida del puntero más allá de la duración (de la función), utilice la primera opción. Si no le preocupa que el destinatario extienda la duración, páselo por referencia y permita que el destinatario lo copie o no.

- Si tiene que proporcionar el acceso de una función del asistente al puntero subyacente y sabe que la función del asistente solo utilizará el puntero y volverá antes de que la función de llamada vuelva, esa función no necesita compartir la propiedad del puntero subyacente. Solo debe tener acceso al puntero dentro de la duración de `shared_ptr` del llamador. En este caso, es seguro pasar el puntero `shared_ptr` por referencia o pasar el puntero sin formato o una referencia al objeto subyacente. Pasarlo de esta manera proporciona una pequeña ventaja de rendimiento y también puede ayudarle a expresar la intención de la programación.
- A veces, por ejemplo en `std::vector<shared_ptr<T>>`, puede ser necesario pasar cada `shared_ptr` a un cuerpo de expresión lambda o a un objeto de función con nombre. Si la expresión lambda o la función no almacena el puntero, debe pasar el puntero `shared_ptr` por referencia para evitar llamar al constructor de copias para cada elemento.

Ejemplo 6

En el ejemplo siguiente se muestra cómo `shared_ptr` sobrecarga distintos operadores de comparación para habilitar las comparaciones de punteros en la memoria que pertenece a las instancias de `shared_ptr`.

```
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;
```

Consulte también

[Punteros inteligentes \(C++ moderno\)](#)

Procedimiento Creación y uso de instancias `weak_ptr`

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

A veces, un objeto debe almacenar una manera de tener acceso al objeto subyacente de una instancia de `shared_ptr` sin hacer que aumente el recuento de referencias.

Normalmente, esta situación se produce cuando se tienen referencias cíclicas entre instancias de `shared_ptr`.

El mejor diseño es evitar la propiedad compartida de punteros siempre que sea posible. Sin embargo, si debe tener la propiedad compartida de las instancias de `shared_ptr`, evite las referencias cíclicas entre ellas. Cuando las referencias cíclicas sean inevitables o incluso preferibles por alguna razón, use `weak_ptr` para dar a uno o varios de los propietarios una referencia débil a otra instancia de `shared_ptr`. Mediante una instancia de `weak_ptr`, puede crear una instancia de `shared_ptr` que se une a un conjunto existente de instancias relacionadas, pero solo si el recurso de memoria subyacente sigue siendo válido. Una instancia de `weak_ptr` propiamente dicha no participa en el recuento de referencias y, por lo tanto, no puede impedir que el recuento de referencias vaya a cero. Sin embargo, puede usar una instancia de `weak_ptr` para intentar obtener una nueva copia de `shared_ptr` con la que se inicializó. Si ya se ha eliminado la memoria, el operador `bool` de `weak_ptr` devuelve `false`. Si la memoria sigue siendo válida, el nuevo puntero compartido incrementa el recuento de referencias y garantiza que la memoria será válida siempre y cuando la variable `shared_ptr` permanezca en el ámbito.

Ejemplo

En el ejemplo de código siguiente se muestra un caso donde se usa `weak_ptr` para garantizar la eliminación adecuada de los objetos que tienen dependencias circulares. Cuando examine el ejemplo, suponga que se creó solo después de que se consideraran soluciones alternativas. Los objetos `Controller` representan algún aspecto de un proceso de máquina y funcionan de forma independiente. Cada controlador debe poder consultar el estado de los demás controladores en cualquier momento y cada uno contiene una instancia de `vector<weak_ptr<Controller>>` privada para este fin. Cada vector contiene una referencia circular y, por lo tanto, se usan instancias de `weak_ptr` en lugar de `shared_ptr`.

```

#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
```

Como experimento, modifique el vector `others` para que sea una instancia de `vector<shared_ptr<Controller>>` y, luego, en la salida, observe que no se invoca ningún destructor cuando `RunTest` devuelve resultados.

Consulte también

[Punteros inteligentes \(C++ moderno\)](#)

Procedimiento Creación y uso de instancias CComPtr y CComQIPtr

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

En la programación clásica de Windows, a menudo las bibliotecas se implementan como objetos COM (o más concretamente, como servidores COM). Muchos componentes del sistema operativo Windows se implementan como servidores COM y muchos colaboradores proporcionan bibliotecas en este formato. Para obtener información sobre los conceptos básicos de COM, vea [Component Object Model \(COM\)](#).

Cuando cree instancias de un objeto del Modelo de objetos componentes (COM), almacena el puntero de interfaz en un puntero inteligente COM, que realiza el recuento de referencias mediante llamadas a `AddRef` y `Release` en el destructor. Si usa Active Template Library (ATL) o la biblioteca MFC (Microsoft Foundation Class), use el puntero inteligente de `CComPtr`. Si no usa ATL o MFC, use `_com_ptr_t`. Como no existe un equivalente COM para `std::unique_ptr`, use estos punteros inteligentes para los escenarios de un solo propietario y de varios propietarios. `CComPtr` y `ComQIPtr` admiten las operaciones de movimiento que tienen referencias `rvalue`.

Ejemplo: CComPtr

En el ejemplo siguiente se muestra cómo usar `CComPtr` para crear una instancia de un objeto COM y obtener punteros a sus interfaces. Observe que la función miembro `CComPtr::CoCreateInstance` se usa para crear el objeto COM, en lugar de la función Win32 que tiene el mismo nombre.

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< IMFMediaControl> pControl;
    hr = pGraph->QueryInterface(IID_PPV_ARGS(&pControl));
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Obtain a third interface.
    CComPtr< IMFMediaEvent> 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` y sus relativos forman parte de la ATL y se definen en `<atlcomcli.h>`. `_com_ptr_t` se declara en `<comip.h>`. El compilador crea especializaciones de `_com_ptr_t` cuando genera clases contenedoras para bibliotecas de tipos.

Ejemplo: `CComQIPtr`

ATL también proporciona `CComQIPtr`, que tiene una sintaxis más sencilla para consultar un objeto COM para recuperar una interfaz adicional. Sin embargo, se recomienda `CComPtr` porque hace todo lo que `CComQIPtr` puede hacer y es semánticamente más coherente con los punteros de interfaz COM sin formato. Si usa un `CComPtr` para consultar una interfaz, el nuevo puntero de interfaz se coloca en un parámetro de salida. Si se produce un error en la llamada, se devuelve un valor HRESULT, que es el patrón COM típico. Con `CComQIPtr`, el valor devuelto es el propio puntero y, si se produce un error en la llamada, no se puede tener acceso al valor devuelto HRESULT interno. Las dos líneas siguientes muestran cómo los mecanismos de control de errores en `CComPtr` y `CComQIPtr` son diferentes.

```

// 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*/ }

```

Ejemplo: IDispatch

`CComPtr` proporciona una especialización para `IDispatch` que le permite almacenar punteros en los componentes de automatización COM e invocar los métodos de la interfaz mediante un enlace en tiempo de ejecución. `CComDispatchDriver` es un `typedef` para `CComQIPtr<IDispatch, &IID_IDispatch>`, que es implícitamente convertible a `CComPtr<IDispatch>`. Por lo tanto, cuando cualquiera de estos tres nombres aparece en el código, es equivalente a `CComPtr<IDispatch>`. En el ejemplo siguiente se muestra cómo obtener un puntero para el modelo de objetos de Microsoft Word mediante un `CComPtr<IDispatch>`.

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();
}
```

Consulte también

[Punteros inteligentes \(C++ moderno\)](#)

Control de excepciones en MSVC

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Una excepción es una condición de error, posiblemente fuera del control del programa, que evita que el programa continúe a lo largo de su ruta normal de ejecución. Ciertas operaciones, incluidas la creación de objetos, la entrada/salida de archivo y las llamadas de función realizadas desde otros módulos, son fuentes potenciales de excepciones, aunque el programa se ejecute correctamente. Un código robusto prevé y controla las excepciones. Para detectar errores lógicos, deben usarse aserciones en lugar de excepciones (consulte [Uso de aserciones](#)).

Tipos de excepciones

El compilador de Microsoft C++ (MSVC) admite tres tipos de control de excepciones:

- [Control de excepciones de C++](#)

Para la mayoría de los programas de C++, debe usarse el control de excepciones de C++. Tiene seguridad de tipos y garantiza que se invoquen los destructores de objetos durante el desenredo de la pila.

- [Control de excepciones estructurado](#)

Windows proporciona su propio mecanismo de excepción, denominado control de excepciones estructurado (SEH). No se recomienda para la programación con C++ o con MFC. SEH solo debe utilizarse en programas que no son de MFC C.

- [Excepciones de MFC](#)

A partir de la versión 3.0, MFC usa excepciones de C++. Aun así, sigue admitiendo sus anteriores macros de control de excepciones, cuya forma es similar a la de las excepciones de C++. Si busca asesoramiento sobre la mezcla de macros de MFC y excepciones de C++, consulte [Excepciones: uso de macros de MFC y excepciones de C++](#).

Use una opción del compilador [/EH](#) para especificar el modelo de control de excepciones que se usará en un proyecto de C++. El control de excepciones de C++ estándar ([/EHsc](#)) es el valor predeterminado en los nuevos proyectos de C++ en Visual Studio.

No se recomienda mezclar mecanismos de control de excepciones. Por ejemplo, no use excepciones de C++ con el control de excepciones estructurado. Si usa exclusivamente

el control de excepciones de C++, el código será más portable y podrá controlar todo tipo de excepciones. Para obtener más información sobre los inconvenientes del control de excepciones estructurado, consulte [Control de excepciones estructurado](#).

En esta sección

- Procedimientos recomendados de C++ moderno para las excepciones y el control de errores
- Cómo: Diseñar para la seguridad de las excepciones
- Cómo: Interfaz entre código excepcional y no excepcional
- Las instrucciones try, catch y throw
- Cómo se evalúan los bloques Catch
- Excepciones y desenredo de la pila
- Especificaciones de excepciones
- noexcept
- Excepciones de C++ no controladas
- Mezclar excepciones de C (estructuradas) y de C++
- Control de excepciones estructurado (C/C++)

Consulte también

[Referencia del lenguaje C++](#)

[Control de excepciones x64](#)

[Control de excepciones \(C++/CLI y C++/CX\)](#)

Procedimientos recomendados de C++ moderno para el control de errores y excepciones

Artículo • 03/03/2023 • Tiempo de lectura: 7 minutos

En C++ moderno, en la mayoría de los escenarios, la manera preferida de notificar y controlar los errores lógicos y los errores en tiempo de ejecución es usar excepciones. Esto es especialmente cierto cuando la pila puede contener varias llamadas a función entre la función que detecta el error y la función que tiene el contexto para controlar el error. Las excepciones proporcionan una forma formal y bien definida para que el código que detecta errores pase la información a la pila de llamadas.

Uso de excepciones para código excepcional

Los errores de programa a menudo se dividen en dos categorías: errores lógicos causados por errores de programación, por ejemplo, un error "índice fuera del intervalo". Y, errores en tiempo de ejecución que están fuera del control del programador, por ejemplo, un error "servicio de red no disponible". En la programación de estilo C y en COM, los informes de errores se administran devolviendo un valor que representa un código de error o un código de estado para una función determinada, o estableciendo una variable global que el autor de la llamada puede recuperar opcionalmente después de cada llamada a función para ver si se han notificado errores. Por ejemplo, la programación COM usa el valor devuelto HRESULT para comunicar errores al autor de la llamada. Y la API Win32 tiene la función `GetLastError` para recuperar el último error notificado por la pila de llamadas. En ambos casos, depende del autor de la llamada reconocer el código y responder a él adecuadamente. Si el autor de la llamada no controla explícitamente el código de error, el programa podría bloquearse sin advertencia. O bien, podría seguir ejecutándose con datos incorrectos y generar resultados incorrectos.

Las excepciones se prefieren en C++ moderno por los siguientes motivos:

- Una excepción obliga al código que llama a reconocer una condición de error y controlarla. Las excepciones no controladas detienen la ejecución del programa.
- Una excepción salta al punto de la pila de llamadas que puede controlar el error. Las funciones intermedias pueden dejar que la excepción se propague. No tienen que coordinarse con otras capas.

- El mecanismo de desenredo de la pila de excepciones destruye todos los objetos del ámbito después de iniciar una excepción, según reglas bien definidas.
- Una excepción permite una separación limpia entre el código que detecta el error y el código que lo controla.

En el ejemplo simplificado siguiente se muestra la sintaxis necesaria para iniciar y detectar excepciones en C++.

```
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;
}
```

Las excepciones en C++ se asemejan a las de lenguajes como C# y Java. En el bloque `try`, si se produce una excepción, la detectará el primer bloque `catch` asociado cuyo tipo coincida con el de la excepción. En otras palabras, la ejecución salta de la instrucción `throw` a la instrucción `catch`. Si no se encuentra ningún bloque `catch` utilizable, se invoca `std::terminate` y se cierra el programa. En C++, se puede producir cualquier tipo; sin embargo, se recomienda iniciar un tipo que derive directa o indirectamente de `std::exception`. En el ejemplo anterior, el tipo de excepción, `invalid_argument`, se define en la biblioteca estándar del archivo de encabezado

`<stdexcept>`. C++ no proporciona ni requiere un bloque `finally` para asegurarse de que todos los recursos se liberan si se produce una excepción. La adquisición de recursos es la expresión de inicialización (RAII), que usa punteros inteligentes, que proporciona la funcionalidad necesaria para la limpieza de recursos. Para más información, consulte [Diseño de la seguridad de excepciones](#). Para información sobre el mecanismo de desenredo de pila de C++, consulte [Excepciones y desenredo de pila](#).

Directrices básicas

El control sólido de errores es complicado en cualquier lenguaje de programación. Aunque las excepciones proporcionan varias características que permiten un buen control de errores, no pueden hacer todo el trabajo por usted. Para aprovechar las ventajas del mecanismo de excepciones, tenga en cuenta las excepciones al diseñar su código.

- Use aserciones para comprobar si hay errores que nunca deben producirse. Use excepciones para comprobar si hay errores que pueden producirse, por ejemplo, errores en la validación de entrada en parámetros de funciones públicas. Para más información, consulte la sección [Excepciones frente a aserciones](#).
- Use excepciones cuando el código que controla el error esté separado del código que detecta el error mediante una o varias llamadas a función intermedias. Considere si usar códigos de error en su lugar en bucles críticos para el rendimiento, cuando el código que controla el error esté estrechamente acoplado al código que lo detecta.
- Para cada función que pueda producir o propagar una excepción, proporcione una de las tres garantías de excepción: la garantía fuerte, la garantía básica o la garantía de que no haya excepciones (`noexcept`). Para más información, consulte [Diseño de la seguridad de excepciones](#).
- Inicie excepciones por valor y detéctelas por referencia. No detectes lo que no puedes manejar.
- No use especificaciones de excepciones, que están en desuso en C++11. Para más información, consulte la sección [Especificaciones de excepciones y `noexcept`](#).
- Use los tipos de excepción de biblioteca estándar cuando se apliquen. Derive tipos de excepción personalizados de la jerarquía [exception Class](#).
- No permita que las excepciones escapen de destructores o funciones de desasignación de memoria.

Excepciones y rendimiento

El mecanismo de excepción tiene un costo de rendimiento mínimo si no se produce ninguna excepción. Si se produce una excepción, el costo del recorrido y el desenredo de pila es comparable aproximadamente al costo de una llamada a función. Se requieren estructuras de datos adicionales para realizar un seguimiento de la pila de llamadas después de especificar un bloque `try`, y se requieren instrucciones adicionales para desenredar la pila si se produce una excepción. Sin embargo, en la mayoría de los escenarios, el costo en el rendimiento y la superficie de memoria no es apreciable. Es probable que el efecto adverso de las excepciones en el rendimiento sea apreciable solo en sistemas con limitaciones de memoria. O bien, en bucles críticos para el rendimiento, donde es probable que se produzca un error con regularidad y haya un acoplamiento estricto entre el código para controlarlo y el código que lo notifica. En cualquier caso, es imposible conocer el costo real de las excepciones sin generar perfiles y medir. Incluso en esos casos poco frecuentes en que el costo es significativo, puede sopesarlo con respecto al aumento de la exactitud, la facilidad de mantenimiento y otras ventajas proporcionadas por una directiva de excepciones bien diseñada.

Excepciones frente a aserciones

Las excepciones y las aserciones son dos mecanismos distintos para detectar errores en tiempo de ejecución en un programa. Use instrucciones `assert` para probar las condiciones durante el desarrollo que nunca deben ser verdaderas si todo el código es correcto. No tiene sentido controlar un error de este tipo utilizando una excepción, porque el error indica que hay que arreglar algo en el código. No representa una condición de que el programa tenga que recuperarse en tiempo de ejecución. Un elemento `assert` detiene la ejecución en la instrucción para que pueda inspeccionar el estado del programa en el depurador. Una excepción continúa la ejecución desde el primer controlador `catch` adecuado. Use excepciones para comprobar las condiciones de error que pueden producirse en tiempo de ejecución incluso si el código es correcto, por ejemplo, "archivo no encontrado" o "memoria insuficiente". Las excepciones pueden controlar estas condiciones, incluso si la recuperación simplemente envía un mensaje a un registro y finaliza el programa. Compruebe siempre los argumentos de las funciones públicas mediante excepciones. Incluso si la función no tiene errores, es posible que no tenga control total sobre los argumentos que un usuario podría pasar a ella.

Excepciones de C++ frente a excepciones de SEH de Windows

Los programas de C y C++ pueden usar el mecanismo de control de excepciones estructurado (SEH) en el sistema operativo Windows. Los conceptos de SEH se asemejan a los de las excepciones de C++, excepto que SEH usa las construcciones `_try`, `_except` y `_finally` en lugar de `try` y `catch`. En el compilador de Microsoft C++ (MSVC), se implementan excepciones de C++ para SEH. Sin embargo, al escribir código de C++, use la sintaxis de excepciones de C++.

Para más información, consulte [Control de excepciones estructurado \(C/C++\)](#).

Especificaciones de excepciones y `noexcept`

Las especificaciones de excepciones se introdujeron en C++ como una manera de especificar las excepciones que podría producir una función. Sin embargo, las especificaciones de excepciones demostraron problemas en la práctica y están en desuso en el borrador estándar de C++11. No se recomienda usar especificaciones de excepciones `throw`, salvo para `throw()`, que indica que la función no permite escapar excepciones. Si debe usar especificaciones de excepciones del formato en desuso `throw(type-name)`, la compatibilidad con MSVC es limitada. Para más información, consulte [Especificaciones de excepciones \(throw\)](#). El especificador `noexcept` se introduce en C++11 como alternativa preferida a `throw()`.

Consulte también

[Interfaz entre código excepcional y no excepcional](#)

[Referencia del lenguaje C++](#)

[Biblioteca estándar de C++](#)

Diseño para la seguridad de excepciones

Artículo • 26/09/2022 • Tiempo de lectura: 7 minutos

Una de las ventajas del mecanismo de excepciones es que la ejecución, así como los datos sobre la excepción, saltan directamente de la instrucción que produce la excepción a la primera instrucción catch que la controla. El controlador puede ser cualquier número de niveles en la pila de llamadas. Las funciones a las que se llama entre la instrucción try y la instrucción throw no se requieren para saber nada sobre la excepción que se produce. Sin embargo, tienen que diseñarse de forma que puedan quedar fuera de ámbito “inesperadamente” en cualquier punto donde una excepción pudiera propagarse de arriba a abajo, y lo hagan sin dejar detrás objetos parcialmente creados, memoria perdida o estructuras de datos que están en estado inutilizable.

Técnicas básicas

Una directiva sólida de control de excepciones requiere una reflexión cuidadosa y debe formar parte del proceso de diseño. Por lo general, la mayoría de las excepciones se detectan y se producen en las capas inferiores de un módulo de software, pero estas capas no tienen normalmente suficiente contexto para controlar el error o para exponer un mensaje a los usuarios finales. En las capas centrales, las funciones pueden detectar y volver a producir una excepción cuando tienen que inspeccionar el objeto de excepción o cuando pueden proporcionar información útil adicional para la capa superior que detecta en última instancia la excepción. Una función debe detectar y “pasar” una excepción solo si puede recuperarse completamente de ella. En muchos casos, el comportamiento correcto en las capas centrales consiste en dejar que una excepción se propague hacia arriba en la pila de llamadas. Incluso en la capa superior, puede ser conveniente dejar que una excepción no controlada termine un programa si la excepción hace que quede en un estado en el que no se puede garantizar la corrección.

Independientemente de cómo una función controla una excepción, para ayudar a garantizar que es “segura para excepciones”, debe diseñarse según las reglas básicas siguientes.

Simplificación de las clases de recursos

Cuando la administración de recursos manual se encapsule en las clases, use una clase que no haga nada más que administrar un solo recurso. Al simplificar la clase, se reduce el riesgo de introducir fugas de recursos. Use [punteros inteligentes](#) cuando sea posible,

como se muestra en el ejemplo siguiente. Este ejemplo es deliberadamente artificial y simplista para resaltar las diferencias cuando se utiliza `shared_ptr`.

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) { }  
};
```

Uso de la expresión RAI para administrar recursos

Para que una función sea segura para excepciones, debe garantizar que los objetos que ha asignado mediante `malloc` o `new` se destruyeron y que todos los recursos, por ejemplo, los identificadores de archivos, se cierran o liberan incluso si se produce una excepción. La expresión *Resource Acquisition Is Initialization* (RAI) enlaza la administración de estos recursos con la duración de las variables automáticas. Cuando una función sale del ámbito, ya sea porque vuelve normalmente o debido a una excepción, se invocan los destructores para todas las variables automáticas totalmente implementadas. Un objeto contenedor RAI, por ejemplo, un puntero inteligente, llama a la función de eliminación o cierre adecuada en el destructor. En el código seguro para excepciones, pasar la propiedad de cada recurso inmediatamente a algún tipo de objeto RAI tiene una importancia crítica. Tenga en cuenta que las clases `vector`, `string`, `make_shared`, `fstream` y similares controlan la adquisición del recurso automáticamente. Sin embargo, las construcciones `unique_ptr` y las tradicionales `shared_ptr` son especiales porque la adquisición de recursos la realiza el usuario en lugar del objeto; por consiguiente, se consideran *Resource Release Is Destruction* pero son cuestionables como RAI.

Las tres garantías de excepción

Normalmente, la seguridad para excepciones se explica en términos de tres garantías de excepción que una función puede proporcionar: la *garantía de que no haya error*, la *garantía fuerte* y la *garantía básica*.

Garantía de que no haya error

La garantía de que no haya error (o “ningún throw”) es la mayor garantía que una función puede proporcionar. Indica que la función no producirá una excepción ni permitirá que se propague. Sin embargo, esta garantía no se puede proporcionar confiablemente a menos que (a) se sepa que todas las funciones a las que llama la función tampoco producen ningún error, (b) se sepa que cualquier excepción que se produzca se detectará antes de que llegue a esta función o (c) se sepa cómo detectar y controlar correctamente todas las excepciones que puedan tener acceso a esta función.

La garantía segura y la seguridad básica se basan en la hipótesis de que los destructores no producen ningún error. Todos los contenedores y tipos de la garantía de la biblioteca estándar que los destructores no producen. También hay un requisito inverso: la biblioteca estándar requiere que los tipos definidos por el usuario que se le proporcionan (por ejemplo, como argumentos de plantilla) deben tener destructores no que produzcan excepciones.

Garantía fuerte

La garantía segura establece que, si una función queda fuera de ámbito debido a una excepción, no se perderá memoria y el estado del programa no se modificará. Una función que proporciona una garantía segura es básicamente una transacción que tiene semántica de confirmación o recuperación, es decir, se ejecuta correctamente por completo o no tiene ningún efecto.

Garantía básica

La garantía básica es la más débil de las tres. Sin embargo, podría ser la mejor opción cuando una garantía segura es demasiado costosa en cuanto al uso de memoria o al rendimiento. La garantía básica establece que si se produce una excepción, no se perderá memoria y el objeto aún estará en un estado utilizable aunque los datos se hayan modificado.

Clases seguras para excepciones

Una clase puede ayudar a garantizar su propia seguridad para excepciones aunque la utilicen funciones no seguras, lo que evita que se construya o se destruya parcialmente. Si existe un constructor de clase antes de la finalización, no se crea nunca el objeto ni se llama nunca al destructor. Aunque las variables automáticas que se inicializan antes de la excepción invoquen sus destructores, se perderá la memoria asignada dinámicamente o los recursos no administrados por un puntero inteligente o una variable automática.

Los tipos integrados garantizan todos que no se produzca ningún error y los tipos de la biblioteca estándar admiten la garantía básica como mínimo. Siga estas instrucciones para cualquier tipo definido por el usuario que deba ser seguro para excepciones:

- Utilice punteros inteligentes u otros contenedores de tipo RAII para administrar todos los recursos. Evite la funcionalidad de administración de recursos en el destructor de clase, porque el destructor no se invocará si el constructor produce una excepción. Sin embargo, si la clase es un administrador de recursos dedicado

que controla un único recurso, es aceptable utilizar el destructor para administrar los recursos.

- Comprenda que una excepción producida en un constructor de clase base no se pasará a un constructor de clase derivada. Si desea convertir y volver a producir la excepción de la clase base en un constructor derivado, utilice un bloque try de función.
- Considere si almacenar todo el estado de clase en un miembro de datos que se encapsula en un puntero inteligente, especialmente si una clase tiene un concepto de "inicialización que permite errores". Aunque C++ permite miembros de datos no inicializados, no admite instancias de clase no inicializadas o parcialmente inicializadas. Un constructor debe ejecutarse correctamente o producir un error; no se crea ningún objeto si el constructor no se ejecuta hasta completarse.
- No permita que ninguna excepción se escape del destructor. Un axioma básico de C++ establece que los destructores no deben permitir que una excepción se propague hacia arriba por la pila de llamadas. Si un destructor debe realizar una operación que pueda producir una excepción, debe hacerlo en un bloque try y pasar la excepción. La biblioteca estándar proporciona esta garantía en todos los destructores que define.

Consulte también

[Procedimientos recomendados de C++ moderno para el control de errores y excepciones](#)

[Cómo: Interfaz entre código excepcional y no excepcional](#)

Interacción entre código excepcional y no excepcional

Artículo • 03/03/2023 • Tiempo de lectura: 8 minutos

En este artículo se describe cómo implementar el control de excepciones coherente en el código de C++ y cómo traducir esas excepciones a códigos de error, y viceversa, en los límites de la excepción.

A veces, el código de C++ tiene que interactuar con código que no usa excepciones (código no excepcional). Esta interacción se conoce como *límite de excepción*. Por ejemplo, quizás desee llamar a la función `createFile` de Win32 en el programa de C++. `CreateFile` no produce excepciones. En su lugar establece los códigos de error que pueden recuperarse mediante la función `GetLastError`. Si el programa de C++ no es insignificante, probablemente sea preferible tener una directiva coherente de control de errores basada en excepciones. Y probablemente no sea conveniente abandonar las excepciones solo porque interactúe con código no excepcional. Tampoco le interesa mezclar directivas de error basadas en excepciones y no basadas en excepciones en el código de C++.

Llamada a funciones no excepcionales desde C++

Cuando se llama a una función sin excepciones desde C++, la idea es ajustar esa función en una función de C++ que detecte cualquier error y posiblemente inicie una excepción. Cuando diseñe una función contenedora de este tipo, decida primero qué tipo de garantía de excepción va a proporcionar: sin excepciones, fuerte o básica. En segundo lugar, diseñe la función para liberar correctamente todos los recursos, por ejemplo, los identificadores de archivo, si se produce una excepción. Normalmente, esto significa que usa punteros inteligentes o administradores de recursos similares para poseer los recursos. Para más información sobre las consideraciones de diseño, consulte [Diseño para la seguridad de las excepciones](#).

Ejemplo

En el ejemplo siguiente se muestra que las funciones de C++ que usan internamente las funciones `createFile` y `ReadFile` de Win32 para abrir y leer dos archivos. La clase `File` es un contenedor RAI (Resource Acquisition Is Initialization) para los identificadores de archivo. Su constructor detecta una condición de "archivo no encontrado" y produce

una excepción para propagar el error en la pila de llamadas del ejecutable de C++ (en este ejemplo, la función `main()`). Si se produce una excepción después de que un objeto `File` se haya construido totalmente, el destructor llama automáticamente a `CloseHandle` para liberar el identificador de archivo. (Si lo prefiere, puede usar la clase `CHandle` de Active Template Library (ATL) para este mismo propósito o `unique_ptr` junto con una función de eliminación personalizada). Las funciones que llaman a las API de Win32 y CRT detectan errores y, a continuación, producen excepciones de C++ mediante la función `ThrowLastErrorIf` definida localmente, que a su vez utiliza la clase `Win32Exception`, derivada de la clase `runtime_error`. Todas las funciones de este ejemplo proporcionan una garantía de excepción fuerte; si se produce una excepción en cualquier momento en estas funciones, no se pierden recursos y no se modifica el estado del programa.

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
    }
}

```

Llamada a código excepcional desde código no excepcional

Los programas de C pueden llamar a las funciones de C++ que se declaran como `extern "C"`. El código escrito en diferentes lenguajes puede usar servidores COM de

C++. Al implementar funciones públicas preparadas para excepciones en C++ para que las invoque código sin excepciones, la función de C++ no debe permitir que ninguna excepción se propague de nuevo al llamador. Estos llamadores no tienen ninguna manera de detectar o controlar excepciones de C++. El programa puede finalizar, filtrar recursos o provocar un comportamiento indefinido.

Se recomienda que la función de C++ `extern "C"` detecte específicamente cada excepción que pueda administrar y, si es necesario, convierta la excepción en un código de error que el llamador comprenda. Si no se conocen todas las excepciones posibles, la función de C++ debe tener un bloque `catch(...)` como último controlador. En ese caso, es mejor notificar un error irrecuperable al llamador, porque el programa podría estar en un estado desconocido e irrecuperable.

El siguiente ejemplo muestra una función que asume que cualquier excepción que pueda producirse es una excepción `Win32Exception` o un tipo de excepción derivado de `std::exception`. La función detecta cualquier excepción de estos tipos y propaga la información de error como código de error Win32 al llamador.

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;
}
```

Cuando se convierten excepciones en códigos de error, puede haber un problema: los códigos de error no contienen a menudo la riqueza de información que una excepción

puede almacenar. Para resolver este problema, puede proporcionar un bloque `catch` para cada tipo de excepción concreto que pueda producirse. Además, puede registrarla para anotar los detalles de la excepción antes de que se convierta en un código de error. Este enfoque puede crear código repetitivo si varias funciones usan el mismo conjunto de bloques `catch`. Una buena manera de evitar la repetición de código es la refactorización de esos bloques en una función de utilidad privada que implemente los bloques `try` y `catch`, y que acepte un objeto de función que se invoque en el bloque `try`. En cada función pública, pase el código a la función de utilidad como una expresión lambda.

C++

```
template<typename Func>
bool Win32ExceptionBoundary(Func&& f)
{
    try
    {
        return f();
    }
    catch(Win32Exception& e)
    {
        SetLastErrorCode(e.GetErrorCode());
    }
    catch(const std::exception& e)
    {
        SetLastErrorCode(MY_APPLICATION_GENERAL_ERROR);
    }
    return false;
}
```

En el ejemplo siguiente se muestra cómo escribir la expresión lambda que define el objeto de función (functor). Una expresión lambda suele ser más fácil de leer insertada que el código que llama a un objeto de función con nombre.

C++

```
bool DiffFiles3(const string& file1, const string& file2)
{
    return Win32ExceptionBoundary([&]() -> bool
    {
        File f1(file1);
        File f2(file2);
        if (IsTextFileDiff(f1, f2))
        {
            SetLastErrorCode(MY_APPLICATION_ERROR_FILE_MISMATCH);
            return false;
        }
        return true;
    });
}
```

```
});  
}
```

Para más información sobre las expresiones lambda, consulte [Expresiones lambda](#).

Llamada a código excepcional mediante código no excepcional a partir de código excepcional

Es posible, pero no se recomienda, producir excepciones en código no compatible con excepciones. Por ejemplo, el programa de C++ puede llamar a una biblioteca que use funciones de devolución de llamada que proporcione. En algunas circunstancias, puede producir excepciones de las funciones de devolución de llamada en el código no excepcional que el autor de la llamada original puede controlar. Sin embargo, las circunstancias en las que las excepciones pueden funcionar correctamente son estrictas. Debe compilar el código de biblioteca de una manera que conserve la semántica de desenredo de la pila. El código no compatible con excepciones no puede hacer nada para interceptar la excepción de C++. Además, el código de biblioteca entre el autor de la llamada y la devolución de llamada no puede asignar recursos locales. Por ejemplo, el código que no es compatible con excepciones no puede tener variables locales que apunten a la memoria de montón asignada. Estos recursos se filtran cuando la pila se desenreda.

Los siguientes requisitos deben cumplirse para producir excepciones en código no compatible con excepciones:

- Se debe poder compilar toda la ruta de acceso del código en código no compatible con excepciones mediante `/EHs`.
- No hay recursos asignados localmente que se puedan filtrar cuando se desenreda la pila.
- El código no tiene ningún controlador de excepciones estructurado `_except` que detecte todas las excepciones.

Dado que producir excepciones en código no excepcional es propenso a errores y puede causar problemas de depuración difíciles de resolver, no se recomienda.

Consulte también

[Procedimientos recomendados de C++ moderno para las excepciones y el control de errores](#)

[Diseño para la seguridad de las excepciones](#)

Instrucciones try, throw y catch (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

Para implementar el control de excepciones en C++, se usan las expresiones `try`, `throw` y `catch`.

En primer lugar, se debe usar un bloque `try` para incluir una o más instrucciones que pueden iniciar una excepción.

Una expresión `throw` indica que se ha producido una condición excepcional, a menudo un error, en un bloque `try`. Se puede usar un objeto de cualquier tipo como operando de una expresión `throw`. Normalmente, este objeto se emplea para comunicar información sobre el error. En la mayoría de los casos, se recomienda usar la `std::exception` clase o una de las clases derivadas definidas en la biblioteca estándar. Si uno de ellos no es adecuado, se recomienda衍生 su propia clase de excepción de `std::exception`.

Para controlar las excepciones que se puedan producir, implemente uno o varios bloques `catch` inmediatamente después de un bloque `try`. Cada bloque `catch` especifica el tipo de excepción que puede controlar.

En este ejemplo se muestra un bloque `try` y sus controladores. Suponga que `GetNetworkResource()` adquiere datos a través de una conexión de red y que los dos tipos de excepción son clases definidas por el usuario que derivan de `std::exception`. Observe que las excepciones se detectan en la referencia `const` de la instrucción `catch`. Se recomienda producir excepciones por valor y detectarlas mediante la referencia `const`.

Ejemplo

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");
    // ...
}

```

Comentarios

El código después de la cláusula `try` es la sección de código protegida. La expresión `throw` *inicia* (es decir, produce) una excepción. El bloque de código que hay detrás de la cláusula `catch` es el controlador de excepciones. Este es el controlador que *detecta* la excepción que se produce si los tipos de las expresiones `throw` y `catch` son compatibles. Para obtener una lista de las reglas que rigen la coincidencia de tipos en los bloques `catch`, consulte [Cómo se evalúan los bloques de captura](#). Si la instrucción `catch` especifica puntos suspensivos (...) en lugar de un tipo, el bloque `catch` controla todos los tipos de excepciones. Cuando se compila con la `/EHa` opción , estas pueden incluir excepciones estructuradas de C y excepciones asincrónicas generadas por el sistema o generadas por la aplicación, como la protección de memoria, las infracciones de división por cero y punto flotante. Puesto que los bloques `catch` se procesan por orden de programa para encontrar un tipo coincidente, un controlador de puntos suspensivos debe ser el último controlador del bloque `try` asociado. Use `catch(...)` con precaución; no permita que un programa continúe a menos que el bloque `catch` sepa cómo controlar la excepción específica que se detecta. Normalmente, un bloque `catch(...)` se emplea para registrar errores y realizar limpiezas especiales antes de que se detenga la ejecución de un programa.

Una `throw` expresión que no tiene ningún operando vuelve a iniciar la excepción que se está controlando actualmente. Se recomienda este formulario al volver a iniciar la excepción, ya que conserva la información del tipo polimórfico de la excepción original. Una expresión así únicamente se debe usar en un controlador `catch` o en una función a

la que se llama desde un controlador `catch`. El objeto de excepción que se vuelve a iniciar es el objeto de excepción original, no una copia.

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;
}
```

Consulte también

[Procedimientos C++ recomendados modernos para excepciones y control de errores](#)

[Palabras clave](#)

[Excepciones de C++ no controladas](#)

[_uncaught_exception](#)

Cómo se evalúan los bloques catch (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

C++ permite iniciar excepciones de cualquier tipo, aunque en general se recomienda iniciar tipos derivados de std::exception. Una excepción de C++ se puede detectar mediante un controlador `catch` que especifique el mismo tipo que la excepción, o mediante un controlador que pueda detectar cualquier tipo de excepción.

Si el tipo de excepción que se inicia es una clase, que también tiene una o varias clases base, se puede detectar mediante los controladores que aceptan las clases base del tipo de la excepción, así como referencias a las bases del tipo de la excepción. Observe que, cuando una referencia detecta una excepción, está enlazada al objeto de excepción real que se ha iniciado; si no, es una copia (igual que un argumento de una función).

Cuando se inicia una excepción, se puede detectar mediante los siguientes tipos de controladores `catch`:

- Un controlador que pueda aceptar cualquier tipo (mediante la sintaxis de puntos suspensivos).
- Un controlador que acepte el mismo tipo que el objeto de excepción; dado que es una copia, se omiten los modificadores `const` y `volatile`.
- Un controlador que acepte una referencia al mismo tipo que el objeto de excepción.
- Un controlador que acepte una referencia a una forma `const` o `volatile` del mismo tipo que el objeto de excepción.
- Un controlador que acepte una clase base del mismo tipo que el objeto de excepción; dado que es una copia, se omiten los modificadores `const` y `volatile`. El controlador `catch` para una clase base no debe preceder al controlador `catch` para la clase derivada.
- Un controlador que acepte una referencia a una clase base del mismo tipo que el objeto de excepción.
- Un controlador que acepte una referencia a una forma `const` o `volatile` de una clase base del mismo tipo que el objeto de excepción.

- Un controlador que acepte un puntero al que pueda convertirse un objeto de puntero iniciado mediante las reglas de conversión de puntero estándar.

El orden en que los controladores `catch` aparecen es importante, porque los controladores de un bloque `try` determinado se examinan por orden de aparición. Por ejemplo, es un error colocar el controlador para una clase base antes del controlador para una clase derivada. Una vez encontrado un controlador `catch` coincidente, no se examinan los controladores subsiguientes. Como resultado, un controlador `catch` de puntos suspensivos debe ser el último controlador de su bloque `try`. Por ejemplo:

```
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.  
}
```

En este ejemplo, el controlador `catch` de puntos suspensivos es el único controlador que se examina.

Consulte también

[Procedimientos recomendados de C++ moderno para las excepciones y el control de errores](#)

Excepciones y desenredo de pila en C++

Artículo • 03/03/2023 • Tiempo de lectura: 4 minutos

En el mecanismo de excepciones de C++, el control se mueve desde la instrucción `throw` hasta la primera instrucción `catch` que puede controlar el tipo producido. Cuando se alcanza la instrucción `catch`, todas las variables automáticas que están en el ámbito entre las instrucciones `throw` y `catch` se destruyen en un proceso que se conoce como *desenrollado de la pila*. En el desenredo de la pila, la ejecución se desarrolla del modo siguiente:

1. El control alcanza la instrucción `try` mediante la ejecución de secuencial normal. La sección protegida en el bloque `try` se ejecuta.
2. Si no se produce ninguna excepción durante la ejecución de la sección protegida, las cláusulas `catch` que siguen al bloque `try` no se ejecutan. La ejecución continúa en la instrucción después de la última cláusula `catch` que sigue al bloque `try` asociado.
3. Si se produce una excepción durante la ejecución de la sección protegida o en cualquier rutina a la que la sección protegida llama directa o indirectamente, se crea un objeto de excepción desde el objeto creado por el operando `throw`. (Esto implica que puede haber un constructor de copia) En este punto, el compilador busca una cláusula `catch` en un contexto de ejecución superior que pueda controlar una excepción del tipo que se produce, o un controlador de `catch` que pueda controlar cualquier tipo de excepción. Los controladores de `catch` se examinan en orden de aparición después del bloque `try`. Si no se encuentra ningún controlador adecuado, se examina el siguiente bloque `try` de inclusión dinámica. Este proceso continúa hasta que se examina el bloque `try` de inclusión extremo.
4. Si no se ha encontrado todavía un controlador coincidente, o si se produce una excepción durante el proceso de desenredo pero antes de que el controlador obtenga el control, se llama a la función `terminate` predefinida en tiempo de ejecución. Si se produce una excepción después de que se produzca la excepción pero antes de que empiece el desenredo, se llama a `terminate`.
5. Si se encuentra un controlador de `catch` coincidente y detecta por valor, su parámetro formal se inicializa copiando el objeto de excepción. Si detecta por referencia, el parámetro se inicializa para hacer referencia al objeto de excepción. Una vez inicializado el parámetro formal, comienza el proceso de desenredo de la

pila. Esto implica la destrucción de todos los objetos automáticos que estaban construidos totalmente, pero todavía no destruidos, entre el principio del bloque `try` asociado al controlador de `catch` y el sitio de producción de la excepción. La destrucción se produce en orden inverso al de construcción. Se ejecuta el controlador de `catch` y el programa reanuda la ejecución después del último controlador, es decir, en la primera instrucción o construcción que no es un controlador de `catch`. El control solo puede entrar en un controlador de `catch` a través de una excepción producida, nunca a través de una instrucción `goto` o una etiqueta `case` en una instrucción `switch`.

Ejemplo de desenrollado de pila

En el ejemplo siguiente se muestra cómo se desenreda la pila cuando se produce una excepción. La ejecución del subprocesso salta de la instrucción `throw` de `C` a la instrucción `catch` de `main` y desenreda cada función a lo largo del recorrido. Observe el orden en que se crean los objetos `Dummy` y se destruyen después cuando salen del ámbito. Observe también que ninguna función se completa excepto `main`, que contiene la instrucción `catch`. La función `A` nunca vuelve de la llamada a `B()` y `B` nunca vuelve de la llamada a `C()`. Si quita los comentarios de la definición del puntero de `Dummy` y la correspondiente instrucción `delete`, y ejecuta después el programa, observe que el puntero nunca se elimina. Esto muestra lo que puede suceder cuando las funciones no proporcionan una garantía de excepción. Para obtener más información, vea [How to: Design for Exceptions](#). Si marca como comentario la instrucción `catch`, puede observar lo que sucede cuando un programa finaliza debido a una excepción no controlada.

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.
```

```
*/
```

Especificaciones de excepciones (throw, noexcept) (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 4 minutos

Las especificaciones de excepciones son una característica del lenguaje C++ que indica la intención del programador sobre los tipos de excepciones que puede propagar una función. Puede especificar que una función podría salir o no mediante una excepción con una *especificación de excepciones*. El compilador puede usar esta información para optimizar las llamadas a la función y finalizar el programa si una excepción inesperada escapa de la función.

Antes de C++17 había dos tipos de especificación de excepciones. La *especificación noexcept* era una novedad de C++11. Especifica si el conjunto de posibles excepciones que pueden escapar de la función está vacío. La *especificación de excepciones dinámica*, o especificación `throw(optional_type_list)`, estaba en desuso en C++11 y se quitó en C++17, excepto para `throw()`, que es un alias para `noexcept(true)`. Esta especificación de excepciones se diseñó para proporcionar información de resumen sobre qué excepciones se pueden iniciar desde una función, pero en la práctica resultó ser problemática. La única especificación de excepciones dinámica que resultó ser útil de alguna manera fue la especificación `throw()` incondicional. Por ejemplo, la declaración de función:

C++

```
void MyFunction(int i) throw();
```

indica al compilador que la función no produce ninguna excepción. Aun así, en el modo `/std:c++14`, esto podría provocar un comportamiento indefinido si la función produce una excepción. Por lo tanto, se recomienda usar el operador `noexcept` en lugar del anterior:

C++

```
void MyFunction(int i) noexcept;
```

En la tabla siguiente se resume la implementación de Microsoft C++ de las especificaciones de excepciones:

| Especificación de la excepción | Significado |
|--------------------------------|-------------|
|--------------------------------|-------------|

| Especificación de la excepción | Significado |
|---|---|
| <code>noexcept</code> <code>noexcept(true)</code> <code>throw()</code> | <p>La función no produce ninguna excepción. En el modo <code>/std:c++14</code> (que es el valor predeterminado), <code>noexcept</code> y <code>noexcept(true)</code> son equivalentes. Cuando se produce una excepción desde una función declarada <code>noexcept</code> o <code>noexcept(true)</code>, se invoca <code>std::terminate</code>. Cuando se produce una excepción desde una función declarada como <code>throw()</code> en el modo <code>/std:c++14</code>, el resultado es un comportamiento indefinido. No se invoca ninguna función específica. Se trata de una divergencia del estándar de C++14, que requería que el compilador invocara <code>std::unexpected</code>.</p> <p>Visual Studio 2017, versión 15.5 y posteriores: en el modo <code>/std:c++17</code>, <code>noexcept</code>, <code>noexcept(true)</code> y <code>throw()</code> son equivalentes. En el modo <code>/std:c++17</code>, <code>throw()</code> es un alias para <code>noexcept(true)</code>. En el modo <code>/std:c++17</code> y versiones posteriores, cuando se produce una excepción desde una función declarada con cualquiera de estas especificaciones, se invoca <code>std::terminate</code> según lo requiera el estándar de C++17.</p> |
| <code>noexcept(false)</code> <code>throw(...)</code> Sin especificación | <p>La función puede producir una excepción de cualquier tipo.</p> |
| <code>throw(type)</code> | <p>(C++14 y versiones anteriores) La función puede producir una excepción del tipo <code>type</code>. El compilador acepta la sintaxis, pero la interpreta como <code>noexcept(false)</code>. En el modo <code>/std:c++17</code> y versiones posteriores, el compilador emite la advertencia C5040.</p> |

Si se usa el control de excepciones en una aplicación, debe haber una función en la pila de llamadas que controle las excepciones producidas antes de salir del ámbito externo de una función marcada como `noexcept`, `noexcept(true)` o `throw()`. Si alguna función llamada entre la que produce una excepción y la que controla la excepción se especifica como `noexcept`, `noexcept(true)` (o `throw()` en el modo `/std:c++17`), el programa finaliza cuando la función noexcept propaga la excepción.

El comportamiento de excepción de una función depende de los factores siguientes:

- El modo de compilación estándar del lenguaje que esté establecido.
- Si está compilando la función con C o C++.
- La opción de compilador `/EH` que se use.
- Si ha especificado explícitamente la especificación de la excepción.

Las especificaciones de excepciones explícitas no se permiten en las funciones de C. Se supone que una función de C no inicia excepciones en `/EHsc` y puede producir

excepciones estructuradas en `/EHs`, `/EHa` o `/EHac`.

En la tabla siguiente se resume si una función de C++ podría iniciarse en varias opciones de control de excepciones del compilador:

| Función | <code>/EHsc</code> | <code>/EHs</code> | <code>/EHa</code> | <code>/EHac</code> |
|--|--------------------|-------------------|-------------------|--------------------|
| Función de C++ sin especificación de excepciones | Sí | Sí | Sí | Sí |
| Función de C++ con la especificación de excepciones <code>noexcept</code> , <code>noexcept(true)</code> o <code>throw()</code> | No | No | Sí | Sí |
| Función de C++ con la especificación de excepciones <code>noexcept(false)</code> , <code>throw(...)</code> o <code>throw(type)</code> | Sí | Sí | Sí | Sí |

Ejemplo

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
```

Consulte también

[Instrucciones try, throw y catch \(C++\)](#)

[Procedimientos recomendados de C++ moderno para las excepciones y el control de errores](#)

noexcept (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

C++11: especifica si una función podría generar excepciones.

Sintaxis

noexcept-specifier:

`noexcept`

noexcept-expression

`throw ()`

noexcept-expression:

`noexcept (constant-expression)`

Parámetros

constant-expression

Expresión constante de tipo `bool` que representa si el conjunto de posibles tipos de excepciones está vacío. La versión incondicional es equivalente a `noexcept(true)`.

Comentarios

Un elemento *noexcept-expression* es un tipo de *especificación de excepción*: un sufijo a una declaración de función que representa un conjunto de tipos que podría coincidir con un controlador de excepciones para cualquier excepción que salga de una función.

El operador condicional unario `noexcept(constant_expression)` cuando

constant_expression produce `true`, y su sinónimo incondicional `noexcept`, especifican que el conjunto de posibles tipos de excepciones que pueden salir de una función está vacío. Es decir, la función nunca inicia una excepción y nunca permite que una excepción se propague fuera de su ámbito. El operador `noexcept(constant_expression)` cuando *constant_expression* produce `false` o la ausencia de una especificación de excepción (que no sea para un destructor o una función de desasignación), indica que el conjunto de excepciones posibles que pueden salir de la función es el conjunto de todos los tipos.

Marque una función como `noexcept` solo si todas las funciones a las que llama, directa o indirectamente, también son `noexcept` o `const`. El compilador no comprueba necesariamente cada ruta de acceso al código en busca de excepciones que podrían

ascender hasta una función `noexcept`. Si una excepción sale del ámbito exterior de una función marcada como `noexcept`, `std::terminate` se invoca inmediatamente y no hay ninguna garantía de que se vayan a invocar los destructores de ningún objeto del ámbito. Use `noexcept` en lugar del especificador dinámico de excepciones `throw()`. La *especificación dinámica de excepciones*, o especificación `throw(optional_type_list)`, quedó en desuso en C++11 y se quitó en C++17, excepto para `throw()`, que es un alias de `noexcept(true)`. Se recomienda aplicar `noexcept` a cualquier función que nunca permita que una excepción se propague a la pila de llamadas. Cuando se declara una función `noexcept`, permite que el compilador genere código más eficaz en varios contextos diferentes. Para más información, consulte [Especificaciones de excepciones](#).

Ejemplo

Una plantilla de función que copia su argumento puede declararse `noexcept` en la condición de que el objeto que se va a copiar sea un tipo de datos antiguo sin formato (POD). Este tipo de función podría declararse de este modo:

C++

```
#include <type_traits>

template <typename T>
T copy_object(const T& obj) noexcept(std::is_pod<T>)
{
    // ...
}
```

Consulte también

[Procedimientos recomendados de C++ moderno para el control de errores y excepciones](#)

[Especificaciones de excepciones \(throw, noexcept\)](#)

Excepciones de C++ no controladas

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Si no se puede encontrar un controlador coincidente (o el controlador `catch` de puntos suspensivos) para la excepción actual, se llama a la función predefinida `terminate` en tiempo de ejecución. (También se puede llamar explícitamente a `terminate` en cualquiera de los controladores). La acción predeterminada de `terminate` es llamar a `abort`. Si desea que `terminate` llame a otra función del programa antes de salir de la aplicación, llame a la función `set_terminate` con el nombre de la función que se va a llamar como argumento único. Puede llamar a `set_terminate` en cualquier punto del programa. La rutina `terminate` siempre llama a la última función especificada como argumento para `set_terminate`.

Ejemplo

En el ejemplo siguiente se inicia una excepción `char *`, pero no contiene un controlador designado para detectar excepciones de tipo `char *`. La llamada a `set_terminate` indica a `terminate` que llame a `term_func`.

```
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;
}
```

Output

```
Output
```

```
term_func was called by terminate.
```

La función `term_func` debe finalizar el programa o el subprocesso actual, idealmente mediante una llamada a `exit`. Si no lo hace y, en su lugar, regresa a su llamador, se llama a `abort`.

Consulte también

[Procedimientos recomendados de C++ moderno para las excepciones y el control de errores](#)

Mezclar excepciones de C (estructuradas) y de C++

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Si desea escribir código portable, no se recomienda utilizar el control de excepciones estructuradas (SEH) en un programa de C++. Sin embargo, puede que a veces desee compilar con [/EHs](#) y mezclar excepciones estructuradas y código fuente de C++ estructurado, con lo que necesitará alguna capacidad para controlar ambos tipos de excepciones. Dado que un controlador de excepciones estructuradas no tiene el concepto de objetos ni de excepciones con tipo, no puede controlar las excepciones producidas por código de C++. Sin embargo, los controladores de C++ `catch` pueden controlar excepciones estructuradas. La sintaxis de control de excepciones de C++ (`try`, `throw`, `catch`) no es aceptada por el compilador de C, pero la sintaxis del control de excepciones estructuradas (`_try`, `_except`, `_finally`) es admitida por el compilador de C++.

Consulte [_set_se_translator](#) para obtener información sobre el control de excepciones estructuradas como excepciones de C++.

Si combina excepciones estructuradas y de C++, tenga en cuenta estos posibles problemas:

- Las excepciones de C++ y las excepciones estructuradas no se pueden mezclar dentro de la misma función.
- Los controladores de finalización (bloques `_finally`) se ejecutan siempre, incluso durante el desenredo después de producirse una excepción.
- El control de excepciones de C++ puede detectar y conservar la semántica de desenredo en todos los módulos compilados con las opciones del compilador [/EH](#), que habilita la semántica de desenredo.
- Puede que haya situaciones en las que no se llame a las funciones de destructor para todos los objetos. Por ejemplo, se podría producir una excepción estructurada al intentar realizar una llamada de función a través de un puntero de función no inicializado. Si los parámetros de función son objetos construidos antes de la llamada, no se llama a los destructores de esos objetos durante el desenredado de la pila.

Pasos siguientes

- [Uso de setjmp o longjmp en programas de C++](#)

Vea más información sobre el uso de `setjmp` y `longjmp` en programas de C++.

- [Controlar excepciones estructuradas en C++](#)

Vea ejemplos de las formas en que puede usar C++ para controlar las excepciones estructuradas.

Consulte también

[Procedimientos recomendados de C++ moderno para las excepciones y el control de errores](#)

Uso de `setjmp` y `longjmp`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Cuando `setjmp` y `longjmp` se usan conjuntamente, proporcionan una manera de ejecutar una instrucción `goto` no local. Normalmente se usan en código de C para pasar el control de la ejecución al control de errores o al código de recuperación en una rutina invocada anteriormente sin usar las convenciones estándar de llamada o devolución.

⊗ Precaución

Dado `setjmp` que y `longjmp` no admiten la destrucción correcta de objetos de marco de pila de forma portable entre compiladores de C++, y dado que podrían degradar el rendimiento al impedir la optimización en variables locales, no se recomienda su uso en programas de C++. Se recomienda usar construcciones `try` y `catch` en su lugar.

Si decide usar `setjmp` y `longjmp` en un programa de C++, incluya también `<setjmp.h>` o `<setjmpex.h>` para garantizar la interacción correcta entre las funciones y el control de excepciones estructurado (SEH) o el control de excepciones de C++.

Específicos de Microsoft

Si usa una opción `/EH` para compilar código de C++, se llama a los destructores para objetos locales durante el desenredo de la pila. Pero si usa `/EHs` o `/EHsc` para compilar y una de las funciones que usa `noexcept` llama a `longjmp`, es posible que no se produzca el desenredo del destructor para esa función, según el estado del optimizador.

En código portable, cuando se ejecuta una llamada a `longjmp`, el estándar no garantiza explícitamente la destrucción correcta de objetos basados en marcos y es posible que otros compiladores no la admitan. Para que se haga una idea, en el nivel de advertencia 4, una llamada a `setjmp` provoca la advertencia C4611, que indica que la interacción entre `_setjmp` y la destrucción de objetos de C++ no es portable.

FIN de Específicos de Microsoft

Consulte también

[Mezclar excepciones de C \(estructuradas\) y de C++](#)

Controlar excepciones estructuradas en C++

Artículo • 03/03/2023 • Tiempo de lectura: 4 minutos

La diferencia principal entre el control de excepciones estructuradas en C (SEH) y el control de excepciones de C++ consiste en que el modelo de control de excepciones de C++ trata con tipos, mientras que el modelo de control de excepciones estructurado de C trata las excepciones de un tipo, específicamente, `unsigned int`. Es decir, las excepciones de C se identifican mediante un valor entero sin signo, mientras que las excepciones de C++ se identifican mediante el tipo de datos. Cuando se produce una excepción estructurada en C, cada controlador posible ejecuta un filtro que examina el contexto de la excepción de C y determina si debe aceptar la excepción, pasársela a otro controlador o ignorarla. Cuando se produce una excepción en C++, puede ser de cualquier tipo.

Una segunda diferencia se debe a que el modelo de control de excepciones estructuradas de C se denomina *asincrónico* porque las excepciones se producen de forma secundaria al flujo de control normal. El mecanismo de control de excepciones de C++ es totalmente *sincrónico*, lo que significa que las excepciones aparecen solo cuando se producen.

Cuando se usa la opción del compilador `/EHs` o `/EHsc`, ningún controlador de excepciones de C++ controla las excepciones estructuradas. Estas excepciones solo se controlan mediante `_except` controladores de excepciones estructuradas o `_finally` controladores de terminación estructurada. Para obtener más información, consulte [Control de excepciones estructuradas \(C/C++\)](#).

Bajo una opción del compilador `/EHa`, si se produce una excepción de C en un programa de C++, puede controlarse mediante un controlador de excepciones estructuradas con su filtro asociado o mediante un controlador `catch` de C++, lo que sea dinámicamente más próximo al contexto de la excepción. Por ejemplo, este programa de C++ de muestra genera una excepción de C dentro de un contexto `try` de C++:

Ejemplo: catch de una excepción de C en un bloque catch de 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.

```

Clases contenedoras de excepciones de C

En un ejemplo sencillo como el anterior, la excepción de C solamente se puede detectar con un controlador `catch` de puntos suspensivos (...). No se comunica al controlador ninguna información sobre el tipo o la naturaleza de la excepción. Aunque este método funcione, en algunos casos puede ser necesario definir una transformación entre los dos modelos de control de excepciones de modo que cada excepción de C se asocie con una clase concreta. Para transformar una, se puede definir una clase "contenedora" de excepciones de C, que se puede utilizar o de la que se puede derivar para atribuir un tipo de clase concreto a una excepción de C. Al hacerlo, cada excepción de C se puede controlar por separado mediante un controlador de C++ `catch` específico, en lugar de que todos ellos se controlen con un único controlador.

La clase contenedora puede tener una interfaz que se compone de algunas funciones miembro que determinan el valor de la excepción y que tienen acceso a la información

extendida del contexto de la excepción proporcionada por el modelo de excepciones de C. Es posible que también desee definir un constructor predeterminado y un constructor que acepte un argumento `unsigned int` (para proporcionar la representación de la excepción de C subyacente) y un constructor de copias bit a bit. A continuación se muestra una posible implementación de la clase contenedora de excepciones de C:

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;
    }
};
```

Para utilizar esta clase, se debe instalar una función personalizada de traducción de excepciones de C a la que llama el mecanismo de control de excepciones internas cada vez que se produce una excepción de C. Dentro de la función de traducción, puede producirse cualquier excepción con tipo (quizás un tipo `SE_Exception` o un tipo de clase derivada de `SE_Exception`) que se puede detectar mediante un controlador `catch` coincidente adecuado de C++. La función de traducción simplemente puede volver, lo que indica que no controló la excepción. Si la propia función de traducción provoca una excepción de C, se llama a `terminate`.

Para especificar una función de traducción personalizada, llame a la función `_set_se_translator` con el nombre de la función de traducción como su único argumento. La función de traductor que escriba se llama una vez para cada invocación a la función en la pila que tenga bloques `try`. No hay ninguna función de traducción predeterminada; si no se especifica una mediante una llamada a `_set_se_translator`, la excepción de C solo se puede detectar con un controlador `catch` de puntos suspensivos.

Ejemplo: uso de una función de traducción personalizada

Por ejemplo, el código siguiente instala una función de traducción personalizada y, a continuación, provoca una excepción de C que se ajusta mediante la clase `SE_Exception`:

C++

```
// exceptions_Exception_Handling_Differences3.cpp
// compile with: /EHa
#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
```

Consulte también

[Mezclar excepciones de C \(estructuradas\) y de C++](#)

Structured Exception Handling (C/C++)

Artículo • 03/03/2023 • Tiempo de lectura: 4 minutos

El control de excepciones estructurado (SEH) es una extensión de Microsoft para C y C++ para controlar determinadas situaciones de código excepcionales, como errores de hardware, correctamente. Aunque Windows y Microsoft C++ admiten SEH, se recomienda usar el control de excepciones de C++ estándar ISO en código de C++. Hace que el código sea más portable y flexible. Sin embargo, para mantener el código existente o en determinadas clases de programas, quizás tenga que seguir utilizando SEH.

Específico de Microsoft:

Gramática

```
try-except-statement :  
    __try compound-statement __except ( filter-expression ) compound-statement  
  
try-finally-statement :  
    __try compound-statement __finally compound-statement
```

Comentarios

Con SEH, puede asegurarse de que los recursos, como los bloques de memoria y los archivos, se liberen correctamente si la ejecución termina inesperadamente. También se pueden controlar determinados problemas, como memoria insuficiente, mediante código estructurado conciso que no utiliza instrucciones `goto` ni realiza pruebas detalladas de los códigos de retorno.

Las `try-except` instrucciones y `try-finally` a las que se hace referencia en este artículo son extensiones de Microsoft para los lenguajes C y C++. Admiten SEH al permitir que las aplicaciones tomen el control de un programa después de que se produzcan eventos que de lo contrario finalizarían la ejecución. Aunque SEH funciona con archivos de código fuente de C++, no está diseñado específicamente para C++. Si usa SEH en un programa de C++ que se compila mediante la `/EHs` opción o `/EHsc`, se llama a los destructores para objetos locales, pero es posible que otro comportamiento de ejecución no sea lo que espera. Para más información, consulte el ejemplo más adelante en este artículo. En la mayoría de los casos, en lugar de SEH, se recomienda usar el control de excepciones de C++ estándar ISO. Mediante el control de excepciones de

C++, puede asegurarse de que el código sea más portátil y puede controlar excepciones de cualquier tipo.

Si tiene código C que usa SEH, puede mezclarlo con código de C++ que use el control de excepciones de C++. Para más información, consulte [Control de excepciones estructuradas en C++](#).

Existen dos mecanismos de SEH:

- [Controladores de excepciones](#), o `_except` bloques, que pueden responder o descartar la excepción en función del `filter-expression` valor. Para obtener más información, consulte [try-except la instrucción](#) .
- [Controladores de terminación](#) o bloques `_finally`, a los que siempre se llama, si una excepción provoca la terminación o no. Para obtener más información, consulte [try-finally la instrucción](#) .

Estas dos clases de controladores son distintas, pero están estrechamente relacionadas mediante un proceso conocido como *desenredo de la pila*. Cuando se produce una excepción estructurada, Windows busca el controlador de excepciones instalado más recientemente que está activo actualmente. El controlador puede hacer una de tres cosas:

- No reconoce la excepción y pasa el control a otros controladores (`EXCEPTION_CONTINUE_SEARCH`).
- Reconocer la excepción pero descartarla (`EXCEPTION_CONTINUE_EXECUTION`).
- Reconocer la excepción y controlarla (`EXCEPTION_EXECUTE_HANDLER`).

El controlador de excepciones que reconoce la excepción puede no estar en la función que se estaba ejecutando cuando se produjo la excepción. Puede estar en una función situada mucho más arriba en la pila. La función que se está ejecutando actualmente y todas las demás funciones del marco de pila finalizan. Durante este proceso, la pila no se *desenlaza*. Es decir, las variables locales no estáticas de las funciones terminadas se borran de la pila.

A medida que se desenreda la pila, el sistema operativo llama a cualquier controlador de terminación que haya escrito para cada función. Mediante un controlador de terminación, se limpian recursos que de lo contrario quedarían abiertos debido a una finalización anormal. Si se ha entrado en una sección crítica, se puede salir de ella en el controlador de terminación. Si el programa se va a cerrar, se pueden realizar otras tareas de mantenimiento como cerrar y quitar archivos temporales.

Pasos siguientes

- Escribir un controlador de excepciones
- Escribir un controlador de finalización
- Controlar excepciones estructuradas en C++

Ejemplo

Como se ha explicado anteriormente, se llama a los destructores de los objetos locales si se utiliza SEH en un programa de C++ y se compila con la opción `/EHa` o `/EHsc`. Sin embargo, el comportamiento durante la ejecución puede no ser el esperado si también se utilizan excepciones de C++. En este ejemplo se muestran estas diferencias de comportamiento.

C++

```
#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;
}
```

Si se utiliza `/EHsc` para compilar este código pero el control de pruebas local `CPPEX` está sin definir, no se ejecuta el destructor `TestClass`. La salida es similar a esta:

Output

```
Triggering SEH exception
Executing SEH __except block
```

Si se utiliza `/EHsc` para compilar el código y `CPPEX` se define mediante `/DCPPEX` (para que se produzca una excepción de C++), se ejecuta el destructor `TestClass` y la salida es similar a la siguiente:

Output

```
Throwing C++ exception
Destroying TestClass!
Executing SEH __except block
```

Si usa `/EHs` para compilar el código, el destructor `TestClass` se ejecuta si se produjo una excepción mediante una expresión `throw` estándar de C++ o mediante SEH. Es decir, si `CPPEX` se define o no. La salida es similar a esta:

Output

```
Throwing C++ exception
Destroying TestClass!
Executing SEH __except block
```

Para más información, consulte [/EH\(Modelo de control de excepciones\)](#).

FIN de Específicos de Microsoft

Consulte también

[Control de excepciones](#)

[Palabras clave](#)

[<exception>](#)

[Control de errores y excepciones](#)

[Control estructurado de excepciones \(Windows\)](#)

Escribir un controlador de excepciones

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Los controladores de excepciones se utilizan normalmente para responder a errores específicos. Puede utilizar la sintaxis del control de excepciones para filtrar todas las excepciones distintas de las que sabe cómo administrar. Las demás excepciones se deben pasar a otros controladores (posiblemente en la biblioteca en tiempo de ejecución o el sistema operativo) creados para buscar esas excepciones concretas.

Los controladores de excepciones utilizan la instrucción try-except.

¿Qué más desea saber?

- [La instrucción try-except](#)
- [Escribir un filtro de excepción](#)
- [Generar excepciones de software](#)
- [Excepciones de hardware](#)
- [Restricciones de los controladores de excepciones](#)

Consulte también

[Structured Exception Handling \(C/C++\)](#)

Instrucción try-except

Artículo • 03/03/2023 • Tiempo de lectura: 5 minutos

La instrucción `try-except` es una extensión específica de Microsoft que admite el manejo estructurado de excepciones en los lenguajes C y C++.

C++

```
// . . .
__try {
    // guarded code
}
__except ( /* filter expression */ ) {
    // termination code
}
// . . .
```

Gramática

`try-except-statement:`

`__try compound-statement __except (expression) compound-statement`

Comentarios

La instrucción `try-except` es una extensión de Microsoft para los lenguajes C y C++.

Permite que las aplicaciones de destino obtengan control cuando se producen eventos que normalmente finalizan la ejecución del programa. Estos eventos se denominan *excepciones estructuradas*, o bien *excepciones* para abreviar. El mecanismo que trata estas excepciones se denomina *control estructurado de excepciones* (SEH).

Para obtener información al respecto, vea la instrucción [try-finally](#).

Las excepciones pueden estar basadas en hardware o software. El control estructurado de excepciones es útil incluso cuando las aplicaciones no se pueden recuperar completamente de las excepciones de hardware o software. SEH permite mostrar información de error y capturar el estado interno de la aplicación para ayudar a diagnosticar el problema. Es especialmente útil para problemas intermitentes que no son fáciles de reproducir.

ⓘ Nota

El control de excepciones estructurado funciona con Win32 para archivos de código fuente de C y C++. Sin embargo, no está diseñado específicamente para C++. Para asegurarse de que el código será más portable, use el control de excepciones de C++. Además, el control de excepciones de C++ es más flexible, ya que puede controlar excepciones de cualquier tipo. Para los programas de C++, le recomendamos que utilice el manejo de excepciones nativo de C++: instrucciones `try`, `catch` y `throw`.

La instrucción compuesta detrás de la cláusula `_try` es el *cuerpo* o la sección *protegida*. La expresión `_except` también se conoce como la expresión de *filtro*. Su valor determina cómo se controla la excepción. La instrucción compuesta detrás de la cláusula `_except` es el controlador de excepción. El controlador especifica las acciones que se deben realizar si se produce una excepción durante la ejecución de la sección body. La ejecución continúa de la siguiente manera:

1. Se ejecuta la sección protegida.
2. Si no se produce ninguna excepción durante la ejecución de la sección protegida, continúa la ejecución de la instrucción después de la cláusula `_except`.
3. Si se produce una excepción durante la ejecución de la sección protegida, o en cualquier rutina que llame la sección protegida, se evalúa la expresión `_except`. Hay tres valores posibles:
 - `EXCEPTION_CONTINUE_EXECUTION` (-1) Se descarta la excepción. La ejecución continúa en el punto donde se ha producido la excepción.
 - `EXCEPTION_CONTINUE_SEARCH` (0) No se reconoce la excepción. La búsqueda de un controlador continúa hacia la parte superior de la pila, primero con las instrucciones `try-except` contenedoras y, después, con los controladores siguientes que tengan mayor prioridad.
 - `EXCEPTION_EXECUTE_HANDLER` (1) La excepción se reconoce. Se transfiere el control al controlador de excepciones mediante la ejecución de la instrucción compuesta `_except`; después, la ejecución continúa tras el bloque `_except`.

La expresión `_except` se evalúa como una expresión de C. Está limitado a un solo valor, el operador de expresión condicional o el operador de coma. Si se requiere un mayor procesamiento, la expresión puede llamar a una rutina que devuelva uno de los tres valores enumerados anteriormente.

Cada aplicación puede tener su propio controlador de excepciones.

No es válido saltar dentro de una instrucción `_try`, pero sí fuera. El controlador de excepciones no se llama si un proceso finaliza en medio de la ejecución de una instrucción `try-except`.

Por compatibilidad con versiones anteriores, `_try`, `_except` y `_leave` son sinónimos de `_try`, `_except` y `_leave` a menos que se especifique la opción del compilador [/Za](#) (Deshabilitar extensiones de lenguaje).

La palabra clave `_leave`.

La palabra clave `_leave` solo es válida dentro de la sección protegida de una instrucción `try-except` y su efecto es saltar al final de la sección protegida. La ejecución de la primera instrucción continúa después del controlador de excepciones.

Una instrucción `goto` también puede saltar fuera de la sección protegida y no degrada el rendimiento como lo hace en una instrucción `try-finally`. Esto se debe a que no se produce el desenredo de la pila. Sin embargo, se recomienda usar la palabra clave `_leave` en lugar de una instrucción `goto`. El motivo es que es menos probable que comete un error de programación si la sección de protección es grande o compleja.

Funciones intrínsecas de control de excepciones estructurado

El manejo estructurado de excepciones proporciona dos funciones intrínsecas que están disponibles para usar con la instrucción `try-except`: [GetExceptionCode](#) y [GetExceptionInformation](#).

`GetExceptionCode` devuelve el código (un entero de 32 bits) de la excepción.

La función intrínseca `GetExceptionInformation` devuelve un puntero a una estructura [EXCEPTION_POINTERS](#) que contiene información adicional sobre la excepción. A través de este puntero, se puede tener acceso al estado que tenía el equipo en el momento de producirse una excepción de hardware. La estructura es como sigue:

C++

```
typedef struct _EXCEPTION_POINTERS {
    PEXCEPTION_RECORD ExceptionRecord;
    PCONTEXT ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

Los tipos de puntero `PEXCEPTION_RECORD` y `PCONTEXT` se definen en el archivo de inclusión `<winnt.h>`, y `_EXCEPTION_RECORD` y `_CONTEXT` se definen en el archivo de inclusión `<excpt.h>`

Puede usar `GetExceptionCode` en el controlador de excepciones. Sin embargo, solo puede usar `GetExceptionInformation` dentro de la expresión de filtro de excepciones. La información a la que apunta suele estar en la pila y ya no está disponible cuando el control se transfiere al controlador de excepciones.

La función intrínseca `AbnormalTermination` está disponible dentro de un controlador de terminación. Devuelve 0 si el cuerpo de la instrucción `try-finally` finaliza secuencialmente. En todos los demás casos, devuelve 1.

`<excpt.h>` define algunos nombres alternativos para estos intrínsecos:

`GetExceptionCode` es equivalente a `_exception_code`

`GetExceptionInformation` es equivalente a `_exception_info`

`AbnormalTermination` es equivalente a `_abnormal_termination`

Ejemplo

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");
}

```

Resultados

Output

```

hello
in try
in try
in filter.
caught AV as expected.
in finally. termination:
    abnormal
in except
world

```

Consulte también

[Escribir un controlador de excepciones](#)
[Structured Exception Handling \(C/C++\)](#)
[Palabras clave](#)

Escribir un filtro de excepción

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Para controlar una excepción puede saltar al nivel del controlador de la excepción o puede continuar la ejecución. En lugar de usar el código de controlador de excepciones para controlar la excepción y pasar por ella, puede usar una expresión *filter* para corregir el problema. Después, con el valor devuelto `EXCEPTION_CONTINUE_EXECUTION` (-1), puede reanudar el flujo normal sin borrar la pila.

ⓘ Nota

Algunas excepciones impiden que continúe el flujo. Si *filter* se evalúa como -1 para una de estas excepciones, el sistema genera una nueva excepción. Si llama a `RaiseException`, puede determinar si la excepción continuará.

Por ejemplo, el código siguiente usa una llamada a una función en la expresión *filter*. Esta función controla el problema y devuelve -1 para reanudar el flujo de control normal:

C++

```
// exceptions_Writing_an_Exception_Filter.cpp
#include <windows.h>
int main() {
    int Eval_Exception( int );
    __try {}
    __except ( Eval_Exception( GetExceptionCode( ) ) ) {
        ;
    }
    void ResetVars( int ) {}
    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
        ResetVars( 0 ); // initializes data to 0
        return EXCEPTION_CONTINUE_EXECUTION;
    }
}
```

Es conveniente usar una llamada a una función en la expresión *filter* siempre que *filter* necesite realizar alguna tarea compleja. La evaluación de la expresión hace que se ejecute la función (en este caso, `Eval_Exception`).

Observe el uso de `GetExceptionCode` para determinar la excepción. Se debe llamar a esta función dentro de la expresión de filtro de la instrucción `_except`. `Eval_Exception` no puede llamar a `GetExceptionCode`, pero se le debe pasar el código de la excepción.

Este controlador pasa el control a otro controlador a menos que la excepción sea un desbordamiento de números enteros o de punto flotante. En tal caso, el controlador llama a una función (`ResetVars` es solo un ejemplo, no una función API) para restablecer algunas variables globales. El bloque de instrucciones `_except`, que en este ejemplo está vacío, no se puede ejecutar nunca porque `Eval_Exception` nunca devuelve `EXCEPTION_EXECUTE_HANDLER (1)`.

El uso de una llamada a una función es una buena técnica de uso general para trabajar con expresiones de filtro complejas. Otras dos características útiles del lenguaje C son:

- El operador condicional
- El operador de coma

El operador condicional suele ser útil aquí. Puede usarse para comprobar un código de retorno concreto y, después, devolver uno de dos valores diferentes. Por ejemplo, el filtro del código siguiente reconoce la excepción solo si esta es

`STATUS_INTEGER_OVERFLOW`:

C++

```
_except( GetExceptionCode() == STATUS_INTEGER_OVERFLOW ? 1 : 0 ) {
```

El propósito del operador condicional en este caso es principalmente proporcionar claridad, ya que el código siguiente genera los mismos resultados:

C++

```
_except( GetExceptionCode() == STATUS_INTEGER_OVERFLOW ) {
```

El operador condicional es más útil en situaciones en las que le interese que el filtro se evalúe como -1, `EXCEPTION_CONTINUE_EXECUTION`.

El operador de coma permite ejecutar varias expresiones en secuencia. Después, devuelve el valor de la última expresión. Por ejemplo, el código siguiente almacena el

código de excepción en una variable y después lo comprueba:

C++

```
__except( nCode = GetExceptionCode(), nCode == STATUS_INTEGER_OVERFLOW )
```

Consulte también

[Escribir un controlador de excepciones](#)

[Structured Exception Handling \(C/C++\)](#)

Generar excepciones de software

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

El sistema no marca como excepciones algunos de los orígenes de errores de programa más comunes. Por ejemplo, si intenta asignar un bloque de memoria pero no hay memoria suficiente, el tiempo de ejecución o la función de API no provoca una excepción, sino que devuelve un código de error.

Sin embargo, puede tratar cualquier condición como excepción detectando esa condición en el código y comunicándolo a continuación mediante una llamada a la función [RaiseException](#). Si marca los errores de esta manera, puede aportar las ventajas del control de excepciones estructurado a cualquier tipo de error en tiempo de ejecución.

Para usar el control de excepciones estructurado con errores:

- Defina su propio código de excepción para el evento.
- Llame `RaiseException` cuando detecte un problema.
- Use filtros de control de excepciones para probar el código de excepción definido.

El archivo <winerror.h> muestra el formato de los códigos de excepción. Para asegurarse de que no define un código en conflicto con un código de excepción existente, establezca el tercer bit más significativo en 1. Los cuatro bits más significativos se deben establecer como se muestra en la tabla siguiente.

| Bits | Valor binario recomendado | Descripción |
|-------------|----------------------------------|--|
| 31- 30 | 11 | Estos dos bits describen el estado básico del código: 11 = error, 00 = correcto, 01 = informativo, 10 = advertencia. |
| 29 | 1 | Bit de cliente. Establézcalo en 1 para los códigos definido por el usuario. |
| 28 | 0 | Bit reservado. (Déjelo establecido en 0). |

Puede establecer los dos primeros bits en un valor distinto del binario 11 si lo desea, aunque el valor de "error" es adecuado para la mayoría de las excepciones. Lo importante es recordar establecer los bits 29 y 28 como se muestra en la tabla anterior.

Por lo tanto, el código de error resultante debe tener los cuatro bits más altos establecidos en E hexadecimal E. Por ejemplo, las definiciones siguientes definen

códigos de excepción que no entran en conflicto con ningún código de excepción de Windows. (Es posible, no obstante, que deba comprobar qué códigos usan los archivos DLL de terceros).

C++

```
#define STATUS_INSUFFICIENT_MEM      0xE0000001
#define STATUS_FILE_BAD_FORMAT        0xE0000002
```

Después de definir un código de excepción, puede usarlo para provocar una excepción. Por ejemplo, el código siguiente provoca la excepción de `STATUS_INSUFFICIENT_MEM` en respuesta a un problema de asignación de memoria:

C++

```
lpstr = _malloc( nBufferSize );
if (lpstr == NULL)
    RaiseException( STATUS_INSUFFICIENT_MEM, 0, 0, 0 );
```

Si desea generar simplemente una excepción, puede establecer los tres últimos parámetros en 0. Los tres últimos parámetros son útiles para pasar información adicional y establecer una marca que evite que los controladores continúen la ejecución. Vea la función [RaiseException](#) en Windows SDK para obtener más información.

En los filtros de control de excepciones, puede probar los códigos que haya definido. Por ejemplo:

C++

```
__try {
    ...
}
__except (GetExceptionCode() == STATUS_INSUFFICIENT_MEM ||
          GetExceptionCode() == STATUS_FILE_BAD_FORMAT )
```

Consulte también

[Escribir un controlador de excepciones](#)

[Control de excepciones estructurado \(C/C++\)](#)

Excepciones de hardware

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

La mayoría de las excepciones estándar reconocidas por el sistema operativo son excepciones definidas por hardware. Windows reconoce algunas excepciones de software de bajo nivel, pero normalmente el sistema operativo las administra mejor.

Windows asigna los errores de hardware de los distintos procesadores a los códigos de excepción de esta sección. En algunos casos, un procesador puede generar solo un subconjunto de estas excepciones. Windows preprocesa la información sobre la excepción y emite el código de excepción correspondiente.

Las excepciones de hardware reconocidas por Windows se resumen en la tabla siguiente:

| Código de excepción | Causa de la excepción |
|-----------------------------------|---|
| STATUS_ACCESS_VIOLATION | Lectura o escritura en una ubicación de memoria inaccesible. |
| STATUS_BREAKPOINT | Detección de un punto de interrupción definido por hardware; se usa solo en depuradores. |
| STATUS_DATATYPE_MISALIGNMENT | Lectura o escritura de datos en una dirección que no está bien alineada; por ejemplo, las entidades de 16 bits se deben alinear en límites de 2 bytes. (No aplicable a procesadores 80x86). |
| STATUS_FLOAT_DIVIDE_BY_ZERO | División del tipo de punto flotante entre 0,0. |
| STATUS_FLOAT_OVERFLOW | Superación del exponente positivo máximo del tipo de punto flotante. |
| STATUS_FLOAT_UNDERFLOW | Superación de la magnitud del exponente negativo menor del tipo de punto flotante. |
| STATUS_FLOATING_RESEVERED_OPERAND | Uso de un formato de punto flotante reservado (uso no válido de formato). |
| STATUS_ILLEGAL_INSTRUCTION | Intento de ejecución de un código de instrucción no definido por el procesador. |
| STATUS_PRIVILEGED_INSTRUCTION | Ejecución de una instrucción no permitida en el modo de equipo actual. |
| STATUS_INTEGER_DIVIDE_BY_ZERO | División de un tipo entero entre 0. |

| Código de excepción | Causa de la excepción |
|-------------------------|---|
| STATUS_INTEGER_OVERFLOW | Intento de operación que supera el intervalo del entero. |
| STATUS_SINGLE_STEP | Ejecución de una instrucción en modo paso a paso; solo se usa en depuradores. |

Depuradores, el sistema operativo u otro código de bajo nivel se ocuparán de administrar muchas de las excepciones que se indican en la tabla anterior. El código no debería administrar errores que no sean de enteros y punto flotante. Por lo tanto, lo normal sería usar el filtro de control de excepciones para omitir las excepciones (que se evalúen como 0). En caso contrario, es posible que los mecanismos de nivel inferior no respondan correctamente. Sin embargo, se pueden tomar las debidas precauciones contra el posible efecto de estos errores de bajo nivel mediante la [escritura de controladores de terminación](#).

Consulte también

[Escribir un controlador de excepciones](#)
[Structured Exception Handling \(C/C++\)](#)

Restricciones de los controladores de excepciones

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

La limitación principal de usar los controladores de excepciones en el código es que no se puede usar una instrucción `goto` para saltar a un bloque de instrucciones `_try`. En su lugar, se debe especificar el bloque de instrucciones a través del flujo de control normal. Puede saltar fuera de un bloque de instrucciones `_try` y anidar los controladores de excepciones a su elección.

Consulte también

[Escribir un controlador de excepciones](#)

[Structured Exception Handling \(C/C++\)](#)

Escribir un controlador de finalización

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

A diferencia de los controladores de excepciones, los controladores de terminación se ejecutan siempre, independientemente de si el bloque de código protegido ha finalizado normalmente. El único propósito del controlador de terminación debe ser garantizar que los recursos, como la memoria, los identificadores y los archivos, se cierran correctamente independientemente de cómo termine de ejecutarse una sección de código.

Los controladores de terminación utilizan la instrucción try-finally.

¿Qué más desea saber?

- [Instrucción try-finally](#)
- [Limpiar recursos](#)
- [Cronología de las acciones en el control de excepciones](#)
- [Restricciones de los controladores de finalización](#)

Consulte también

[Structured Exception Handling \(C/C++\)](#)

Instrucción `try-finally`

Artículo • 03/03/2023 • Tiempo de lectura: 4 minutos

La instrucción `try-finally` es una extensión específica de Microsoft que admite el manejo estructurado de excepciones en los lenguajes C y C++.

Sintaxis

La sintaxis siguiente describe la instrucción `try-finally`:

C++

```
// . . .
__try {
    // guarded code
}
__finally {
    // termination code
}
// . . .
```

Gramática

```
try-finally-statement:
    __try compound-statement __finally compound-statement
```

La instrucción `try-finally` es una extensión de Microsoft a los lenguajes C y C++ que permite que las aplicaciones de destino garanticen la ejecución del código de limpieza cuando se interrumpe la ejecución de un bloque de código. La limpieza consta de tareas como desasignar memoria, cerrar archivos y liberar identificadores de archivo. La instrucción `try-finally` es especialmente útil para las rutinas que tienen varios lugares donde comprobar un error, lo que puede causar que la rutina termine antes de tiempo.

Para obtener información relacionada y un ejemplo de código, consulte la [Instrucción try-except](#). Para obtener más información sobre el control de excepciones estructurado en general, consulte [Control de excepciones estructuradas](#). Para obtener más información sobre cómo se controlan las excepciones en aplicaciones administradas con C++/CLI, consulte [Control de excepciones con /clr](#).

 Nota

El control de excepciones estructurado funciona con Win32 para archivos de código fuente de C y C++. Sin embargo, no está diseñado específicamente para C++. Para asegurarse de que el código será más portable, use el control de excepciones de C++. Además, el control de excepciones de C++ es más flexible, ya que puede controlar excepciones de cualquier tipo. Para los programas de C++, se recomienda usar el mecanismo de control de excepciones de C++ (`try`, `catch` y `throw`).

La instrucción compuesta detrás de la cláusula `_try` es la sección protegida. La instrucción compuesta detrás de la cláusula `_finally` es el controlador de terminación. El controlador especifica un conjunto de acciones que se ejecutan cuando se sale de la sección protegida, ya sea si se sale de esta sección por una excepción (terminación anómala) o después de pasar por ella (terminación normal).

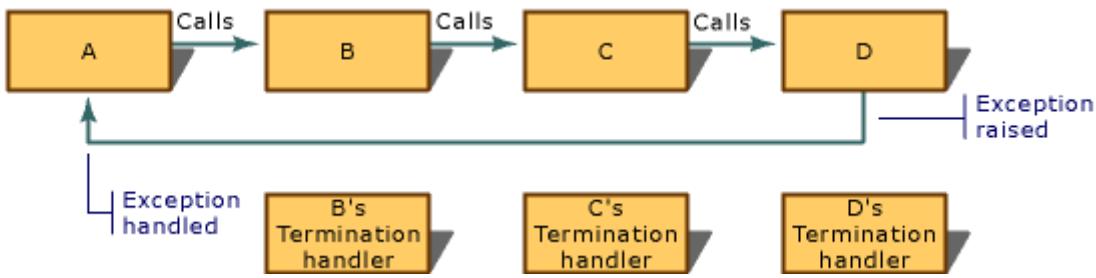
El control llega a una instrucción `_try` mediante la ejecución secuencial simple (paso explícito). Cuando el control entra en `_try`, su controlador asociado se activa. Si el flujo de control alcanza el final del bloque `try`, la ejecución continúa del modo siguiente:

1. Se invoca al controlador de terminación.
2. Cuando el controlador de terminación finaliza, la ejecución continúa después de la instrucción `_finally`. Sin embargo, la sección protegida (por ejemplo, mediante una instrucción `goto` fuera del cuerpo protegido o una instrucción `return`), el controlador de finalización se ejecuta *antes* de que el flujo de control salga de la sección protegida.

Una instrucción `_finally` no bloquea la búsqueda de un controlador de excepciones adecuado.

Si se produce una excepción en el bloque `_try`, el sistema operativo debe buscar un controlador para la excepción; de lo contrario, el programa producirá un error. Si se encuentra el controlador, se ejecutan todos y cada uno de los bloques `_finally` y la ejecución se reanuda en el controlador.

Por ejemplo, suponga que una serie de llamadas de función vincula la función A a la función D, como se muestra en la ilustración siguiente. Cada función tiene un controlador de finalización. Si se produce una excepción en la función D y se controla en A, se llama a los controladores de finalización en este orden mientras el sistema desenreda la pila: D, C, B.



Orden de terminación-ejecución de controladores

ⓘ Nota

El comportamiento de try-finally es diferente del de otros lenguajes que admiten el uso de `finally`, como C#. Una instrucción `_try` puede tener `_finally` o `_except`, pero no ambos. Si se van a usar ambos conjuntamente, una instrucción try-except externa debe incluir la instrucción try-finally interna. Las reglas que especifican cuándo se ejecuta cada bloque también son diferentes.

A efectos de compatibilidad con versiones anteriores, `_try`, `_finally` y `_leave` son sinónimos de `_try`, `_finally` y `_leave` a menos que se especifique la opción del compilador `/Za(Deshabilitar extensiones de lenguaje)`.

La palabra clave `_leave`

La palabra clave `_leave` solo es válida dentro de la sección protegida de una instrucción `try-finally` y su efecto es saltar al final de la sección protegida. La ejecución continúa en la primera instrucción del controlador de finalización.

Una instrucción `goto` también puede saltar fuera de la sección protegida, pero el rendimiento se degrada porque invoca el desenredado de la pila. La instrucción `_leave` es más eficaz porque no produce el desenredado de la pila.

Finalización anómala

La salida de una instrucción `try-finally` mediante el uso de la función en tiempo de ejecución `longjmp` se considera una finalización anómala. No es válido saltar dentro de una instrucción `_try`, pero sí fuera. Todas las instrucciones `_finally` que están activas entre el punto de salida (finalización normal del bloque `_try`) y el punto de destino (el bloque `_except` que controla la excepción) deben ejecutarse. Esto recibe el nombre de *desenredado local*.

Si un bloque `_try` se finaliza de forma prematura por cualquier motivo, incluido un salto fuera del bloque, el sistema ejecuta el bloque `_finally` asociado como parte del proceso de desenredado de la pila. En tales casos, la función `AbnormalTermination` devuelve `true` si se llama desde el bloque `_finally`, de lo contrario, devuelve `false`.

No se llama al controlador de finalización si un proceso se elimina en medio de la ejecución de una instrucción `try-finally`.

FIN de Específicos de Microsoft

Consulte también

[Escribir un controlador de finalización](#)

[Structured Exception Handling \(C/C++\)](#)

[Palabras clave](#)

[Sintaxis del controlador de terminación](#)

Limpiar recursos

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Durante la ejecución del controlador de terminación, tal vez no sepa qué recursos se han adquirido antes de que se llame al controlador de terminación. Es posible que el bloque de instrucciones `__try` se interrumpiera antes de que se adquirieran todos los recursos, de modo que no todos los recursos estuvieran abiertos.

Por tanto, como medida de seguridad, debe comprobar qué recursos están abiertos antes de proceder con la limpieza de controladores de terminación. Un procedimiento recomendado es:

1. Inicializar los identificadores en NULL.
2. En el bloque de instrucciones `__try`, adquiera recursos. Los manipuladores se establecen en valores positivos cuando se adquiere el recurso.
3. En el bloque de instrucciones `__finally`, liberar cada recurso cuyo manipulador o variable de marca correspondiente sea distinto de cero o distinto de NULL.

Ejemplo

Por ejemplo, el código siguiente utiliza un controlador de terminación para cerrar tres archivos y liberar un bloque de memoria. Estos recursos se adquirieron en el bloque de instrucciones `__try`. Antes de limpiar un recurso, el código comprueba primero si el recurso se adquirió.

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();
}
```

Consulte también

[Escribir un controlador de finalización](#)
[Structured Exception Handling \(C/C++\)](#)

Resumen sobre los intervalos de control de excepciones Resumen

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Un controlador de terminación se ejecuta independientemente de cómo finalice el bloque de instrucciones `__try`. La terminación se puede producir cuando se sale del bloque `__try`, cuando una instrucción `longjmp` transfiere el control fuera del bloque y cuando se desenreda la pila debido al control de excepciones.

ⓘ Nota

El compilador de Microsoft C++ admite dos versiones de las instrucciones `setjmp` y `longjmp`. La versión rápida omite el control de terminación pero es más eficaz. Para usar esta versión, incluya el archivo `<setjmp.h>`. La otra versión admite el control de terminación como se describe en el párrafo anterior. Para usar esta versión, incluya el archivo `<setjmpex.h>`. El aumento del rendimiento de la versión rápida depende de la configuración de hardware.

El sistema operativo ejecuta todos los controladores de terminación en el orden adecuado antes de que cualquier otro código pueda ejecutarlos, incluido el cuerpo de un controlador de excepciones.

Cuando la causa de la interrupción es una excepción, el sistema debe ejecutar primero la parte del filtro de uno o varios controladores de excepciones antes de decidir qué debe terminar. El orden de los eventos es:

1. Se produce una excepción.
2. El sistema mira la jerarquía de los controladores de excepciones activos y ejecuta el filtro del controlador con mayor precedencia. Ese es el controlador de excepciones instalado y más profundamente anidado, pasando por bloques y llamadas a función.
3. Si este filtro pasa el control (devuelve 0), el proceso continúa hasta que se encuentra un filtro que no pase el control.
4. Si este filtro devuelve -1, la ejecución continúa donde se produjo la excepción y la terminación no tiene lugar.
5. Si el filtro devuelve 1, se producen los eventos siguientes:

- El sistema desenreda la pila: borra todos los marcos de pila entre el lugar donde se produjo la excepción y el marco de pila que contiene el controlador de excepciones.
- Cuando se desenreda la pila, se ejecuta cada controlador de terminación de la pila.
- Se ejecuta el propio controlador de excepciones.
- El control se pasa a la línea de código después del final de este controlador de excepciones.

Consulte también

[Escribir un controlador de finalización](#)
[Structured Exception Handling \(C/C++\)](#)

Restricciones de los controladores de finalización

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

No puede utilizar una instrucción `goto` para saltar dentro de un bloque de instrucciones `__try` o un bloque de instrucciones `__finally`. En su lugar, se debe especificar el bloque de instrucciones a través del flujo de control normal. (Sin embargo, sí se puede saltar de un bloque de instrucciones `__try`). Además, no puede anidar un controlador de excepciones ni un controlador de finalización dentro de un bloque de instrucciones `__finally`.

Algunos tipos de código permitidos en un controlador de finalización generan resultados cuestionables, por lo que deben utilizarse con precaución, si se utilizan. Uno de ellos es una instrucción `goto` que salta fuera de un bloque de instrucciones `__finally`. Si el bloque se ejecuta como parte de una finalización normal, no sucede nada inusual. Pero si el sistema está desenredando la pila, el desenredado se detiene y la función actual obtiene el control como si no hubiera una finalización anómala.

Una instrucción `return` dentro de un bloque de instrucciones `__finally` presenta casi la misma situación. El control se devuelve al llamador inmediato de la función que contiene el controlador de finalización. Si el sistema estaba desenredando la pila, este proceso se detiene y el programa continúa como si no se hubiera producido una excepción.

Consulte también

[Escribir un controlador de finalización](#)
[Structured Exception Handling \(C/C++\)](#)

Transportar excepciones entre subprocessos

Artículo • 03/03/2023 • Tiempo de lectura: 11 minutos

El compilador de Microsoft C++ (MSVC) admite *transportar una excepción* de un subprocesso a otro. El transporte de excepciones permite detectar una excepción en un subprocesso y hacer que parezca que la excepción se produce en un subprocesso diferente. Por ejemplo, esta característica se puede utilizar para escribir una aplicación multiproceso en la que el subprocesso principal controla todas las excepciones producidas por sus subprocessos secundarios. El transporte de excepciones es útil principalmente para los desarrolladores que crean bibliotecas de programación o sistemas paralelos. Para implementar el transporte de excepciones, MSVC proporciona el tipo `exception_ptr` y las funciones `current_exception`, `rethrow_exception` y `make_exception_ptr`.

Sintaxis

```
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;
}
```

Parámetros

unspecified

Clase interna sin especificar que se utiliza para implementar el tipo `exception_ptr`.

p

Objeto `exception_ptr` que hace referencia a una excepción.

E

Clase que representa una excepción.

e

Instancia de la clase del parámetro `E`.

Valor devuelto

La función `current_exception` devuelve un objeto `exception_ptr` que hace referencia a la excepción que está en curso actualmente. Si no hay ninguna excepción en curso, la función devuelve un objeto `exception_ptr` que no está asociado a ninguna excepción.

La función `make_exception_ptr` devuelve un objeto `exception_ptr` que hace referencia a la excepción especificada por el parámetro `e`.

Comentarios

Escenario

Suponga que desea crear una aplicación que se pueda escalar para controlar una cantidad de trabajo variable. Para lograr este objetivo, diseña una aplicación multiproceso en la que un subprocesso principal inicial crea tantos subprocessos secundarios como necesita para hacer el trabajo. Los subprocessos secundarios ayudan al subprocesso principal a administrar los recursos, equilibrar las cargas y mejorar el rendimiento. Al distribuir el trabajo, la aplicación multiproceso funciona mejor que una aplicación uniproceso.

Sin embargo, si un subprocesso secundario produce una excepción, subprocesso principal debe controlarla. Esto es porque desea que la aplicación controle las excepciones de una manera coherente y unificada independientemente del número de subprocessos secundarios.

Solución

Para controlar el escenario anterior, el estándar de C++ admite transportar una excepción entre subprocessos. Si un subprocesso secundario produce una excepción, esa excepción se convierte en la *excepción actual*. Por analogía con el mundo real, se dice que la excepción actual está *en vuelo*. La excepción actual está en vuelo desde el momento en que se produce hasta que el controlador de excepciones que la captura vuelve.

El subprocesso secundario puede detectar la excepción actual en un bloque `catch` y llamar después a la función `current_exception` para almacenar la excepción en un objeto `exception_ptr`. El objeto `exception_ptr` debe estar disponible para el subprocesso secundario y para el subprocesso principal. Por ejemplo, el objeto `exception_ptr` puede ser una variable global cuyo acceso esté controlado mediante una

exclusión mutua. El término *transportar una excepción* significa que una excepción de un subprocesso se puede convertir a un formato al que puede tener acceso otro subprocesso.

A continuación, el subprocesso principal llama a la función `rethrow_exception`, que extrae y produce la excepción desde el objeto `exception_ptr`. Cuando se produce la excepción, se convierte en la excepción actual del subprocesso principal. Es decir, parece que la excepción procede del subprocesso principal.

Por último, el subprocesso principal puede detectar la excepción actual en un bloque `catch` y después procesarla o producirla en un controlador de excepciones de nivel superior. O bien, el subprocesso principal puede omitir la excepción y permitir que el proceso finalice.

La mayoría de las aplicaciones no tienen que transportar excepciones entre subprocessos. Sin embargo, esta característica es útil en un sistema informático en paralelo porque el sistema puede repartir el trabajo entre los subprocessos secundarios, los procesadores o los núcleos. En un entorno informático en paralelo, un único subprocesso dedicado puede controlar todas las excepciones de los subprocessos secundarios y puede presentar un modelo coherente de control de excepciones a cualquier aplicación.

Para obtener más información sobre la propuesta del comité de normas de C++, busque en Internet el número de documento N2179, titulado "Language Support for Transporting Exceptions between Threads" (Compatibilidad con lenguajes para transportar excepciones entre subprocessos).

Modelos de control de excepciones y opciones del compilador

El modelo de control de excepciones de una aplicación determina si puede detectar y transportar una excepción. Visual C++ admite tres modelos que pueden controlar excepciones de C++, excepciones de Control de excepciones estructurado (SEH) y excepciones de Common Language Runtime (CLR). Use las opciones del compilador `/EH` y `/clr` para especificar el modelo de control de excepciones de la aplicación.

Solo la combinación siguiente de opciones del compilador e instrucciones de programación puede transportar una excepción. Otras combinaciones no pueden detectar excepciones, o pueden detectar pero no pueden transportar excepciones.

- La opción del compilador `/EHa` y la instrucción `catch` pueden transportar excepciones de SEH y C++.

- Las opciones del compilador /EH_a, /EH_s y /EH_{sc}, y la instrucción `catch` pueden transportar excepciones de C++.
- La opción del compilador /CLR y la instrucción `catch` pueden transportar excepciones de C++. La opción del compilador /CLR implica la especificación de la opción /EH_a. Tenga en cuenta que el compilador no admite transportar excepciones administradas. Esto se debe a que las excepciones administradas, que se derivan de la [clase System.Exception](#), ya son objetos que se pueden mover entre subprocessos mediante las funciones de Common Language Runtime.

Importante

Se recomienda que se especifique la opción del compilador /EH_{sc} y detecte solo las excepciones de C++. Se expone a una amenaza de seguridad si utiliza la opción del compilador /EH_a o /CLR y una instrucción `catch` con una *declaración de excepción* de puntos suspensivos (`catch(...)`). Probablemente piense utilizar la instrucción `catch` para capturar algunas excepciones concretas. Sin embargo, la instrucción `catch(...)` captura todas las excepciones de C++ y SEH, incluidas las inesperadas que deben ser irrecuperables. Si se omite o se controla mal una excepción inesperada, el código malintencionado puede aprovechar esa oportunidad para socavar la seguridad del programa.

Uso

En las próximas secciones se describe cómo transportar excepciones mediante el tipo `exception_ptr` y las funciones `current_exception`, `rethrow_exception` y `make_exception_ptr`.

Tipo exception_ptr

Utilice un objeto `exception_ptr` para hacer referencia a la excepción actual o a una instancia de una excepción especificada por el usuario. En la implementación de Microsoft, una excepción se representa mediante una estructura [EXCEPTION_RECORD](#). Cada objeto `exception_ptr` incluye un campo de referencia de excepción que apunta a una copia de la estructura `EXCEPTION_RECORD` que representa la excepción.

Cuando se declara una variable `exception_ptr`, la variable no está asociada a ninguna excepción. Es decir, su campo de referencia de excepción es NULL. Este tipo de objeto

`exception_ptr` se denomina *exception_ptr null*.

Utilice la función `current_exception` o `make_exception_ptr` para asignar una excepción a un objeto `exception_ptr`. Cuando se asigna una excepción a una variable `exception_ptr`, el campo de referencia de excepción de la variable apunta a una copia de la excepción. Si no hay memoria suficiente para copiar la excepción, el campo de referencia de excepción apunta a una copia de una excepción `std::bad_alloc`. Si la función `current_exception` o `make_exception_ptr` no puede copiar la excepción por cualquier otro motivo, la función llama a la función `terminate` para salir del proceso actual.

A pesar de su nombre, un objeto `exception_ptr` no es en sí mismo un puntero. No obedece a la semántica de los punteros y no se puede usar con el operador de acceso del miembro puntero (`->`) o de direccionamiento indirecto (`*`) de miembro de puntero. El objeto `exception_ptr` no tiene ningún miembro de datos ni ninguna función miembro de tipo público.

Comparaciones

Se pueden usar los operadores de igualdad (`==`) y desigualdad (`!=`) para comparar dos objetos `exception_ptr`. Los operadores no comparan el valor binario (patrón de bits) de las estructuras `EXCEPTION_RECORD` que representan las excepciones. En su lugar, los operadores comparan las indicaciones del campo de referencia de excepción de los objetos `exception_ptr`. Por tanto, un `exception_ptr` NULL y el valor NULL se consideran iguales.

Función `current_exception`

Llame a la función `current_exception` en un bloque `catch`. Si una excepción está en vuelo y el bloque `catch` puede detectarla, la función `current_exception` devuelve un objeto `exception_ptr` que hace referencia a la excepción. De lo contrario, la función devuelve un objeto `exception_ptr` NULL.

Detalles

La función `current_exception` captura la excepción que está en vuelo independientemente de si la instrucción `catch` especifica una instrucción de declaración de excepción o no.

Se llama al destructor de la excepción actual al final del bloque `catch` si no vuelve a producir la excepción. En cambio, incluso aunque llame a la función `current_exception` en el destructor, la función devuelve un objeto `exception_ptr` que hace referencia a la excepción actual.

Las llamadas sucesivas a la función `current_exception` devuelven objetos `exception_ptr` que hacen referencia a distintas copias de la excepción actual. Por tanto, al comparar los objetos se consideran diferentes porque hacen referencia a copias distintas, incluso aunque las copias tengan el mismo valor binario.

Excepciones SEH

Si se usa la opción del compilador `/EHs`, se puede detectar una excepción SEH en un bloque `catch` de C++. La función `current_exception` devuelve un objeto `exception_ptr` que hace referencia a la excepción SEH. La función `rethrow_exception` produce la excepción SEH si se le llama con el objeto `exception_ptr` transportado como argumento.

La función `current_exception` devuelve un `exception_ptr` `NULL` si se le llama en un controlador de terminación de SEH `_finally`, un controlador de excepciones `_except` o la expresión de filtro `_except`.

Una excepción transportada no admite excepciones anidadas. Se genera una excepción anidada si se produce otra excepción mientras se está controlando una excepción. Si se detecta una excepción anidada, el miembro de datos `EXCEPTION_RECORD.ExceptionRecord` apunta a una cadena de estructuras `EXCEPTION_RECORD` que describen las excepciones asociadas. La función `current_exception` no admite excepciones anidadas porque devuelve un objeto `exception_ptr` cuyo miembro de datos `ExceptionRecord` se pone a cero.

Si se detecta una excepción SEH, debe administrar la memoria a la que hace referencia cualquier puntero en la matriz de miembros de datos `EXCEPTION_RECORD.ExceptionInformation`. Debe garantizar que la memoria es válida mientras dura el objeto `exception_ptr` correspondiente y que la memoria se liberará cuando se elimine el objeto `exception_ptr`.

Puede utilizar funciones de traductor de excepciones estructuradas (SE) junto con la característica de transporte de excepciones. Si una excepción SEH se traduce a una excepción de C++, la función `current_exception` devuelve un `exception_ptr` que hace referencia a la excepción traducida en lugar de a la excepción SEH original. La función `rethrow_exception` produce posteriormente la excepción traducida, no la excepción

original. Para obtener más información sobre las funciones de traductor de SE, consulte [_set_se_translator](#).

Función rethrow_exception

Después de almacenar una excepción detectada en un objeto `exception_ptr`, el subproceso principal puede procesar el objeto. En el subproceso principal, llame a la función `rethrow_exception` junto con el objeto `exception_ptr` como argumento. La función `rethrow_exception` extrae la excepción del objeto `exception_ptr` y después produce la excepción en el contexto del subproceso principal. Si el parámetro *p* de la función `rethrow_exception` es un `exception_ptr` NULL, la función produce `std::bad_exception`.

La excepción extraída es ahora la excepción actual en el subproceso principal y puede controlarla como haría con cualquier otra excepción. Si se detecta la excepción, puede controlarla inmediatamente o utilizar una instrucción `throw` para enviarla a un controlador de excepciones de nivel superior. De lo contrario, no haga nada y deje que el controlador de excepciones predeterminado del sistema finalice el proceso.

make_exception_ptr (Función)

La función `make_exception_ptr` toma una instancia de una clase como argumento y devuelve un `exception_ptr` que hace referencia a la instancia. Normalmente, se especifica un objeto [exception \(Clase\)](#) como argumento para la función `make_exception_ptr`, aunque el argumento puede ser cualquier objeto de clase.

Llamar a la función `make_exception_ptr` equivale a producir una excepción de C++, detectarla en un bloque `catch` y llamar después a la función `current_exception` para devolver un objeto `exception_ptr` que hace referencia a la excepción. La implementación de Microsoft de la función `make_exception_ptr` es más eficaz que producir y detectar después una excepción.

Una aplicación no suele necesitar la función `make_exception_ptr` y desaconsejamos su uso.

Ejemplo

En el ejemplo siguiente se transporta una excepción estándar de C++ y una excepción personalizada de C++ de un subproceso a otro.

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.

```

Requisitos

Encabezado:<exception>

Consulte también

[Control de excepciones](#)

[/EH \(Modelo de control de excepciones\)](#)

[/clr \(Compilación de Common Language Runtime\)](#)

Aserción y mensajes proporcionados por el usuario (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

El lenguaje C++ admite tres mecanismos de control de errores que ayudan a depurar la aplicación: la [directiva #error](#), la palabra clave [static_assert](#) y la macro [assert Macro, _assert, _wassert](#). Los tres mecanismos emiten mensajes de error y dos de ellos también prueban las aserciones de software. Una asercción de software especifica una condición que se espera que sea cierta (valor true) en un determinado punto del programa. Si se produce un error de asercción en tiempo de compilación, el compilador emite un mensaje de diagnóstico y un error de compilación. Si se produce un error de asercción en tiempo de ejecución, el sistema operativo emite un mensaje de diagnóstico y cierra la aplicación.

Comentarios

La duración de la aplicación se compone de una fase de preprocesamiento, una de compilación y una de tiempo de ejecución. Cada mecanismo de control de errores tiene acceso a la información de depuración disponible durante una de estas fases. Para depurar eficazmente, seleccione el mecanismo que proporciona la información adecuada sobre esa fase:

- La [directiva #error](#) está vigente en el momento del preprocesamiento. Emite incondicionalmente un mensaje definido por el usuario y hace que se produzca un error de compilación. El mensaje puede contener texto que se manipula mediante directivas de preprocesador, pero no se evalúa ninguna expresión resultante.
- La declaración [static_assert](#) está vigente en el momento de la compilación. Prueba una asercción de software que está representada por una expresión de tipo entero definida por el usuario que se puede convertir en un valor booleano. Si la expresión se evalúa como cero (false), el compilador emite el mensaje definido por el usuario y se produce un error de compilación.

La declaración [static_assert](#) resulta especialmente útil para depurar plantillas, porque los argumentos de plantilla se pueden incluir en la expresión especificada por el usuario.

- La macro [assert Macro, _assert, _wassert](#) está vigente en el momento de la ejecución. Evalúa una expresión definida por el usuario y, si el resultado es cero, el sistema emite un mensaje de diagnóstico y cierra la aplicación. Muchas otras

macros, como `_ASSERT` y `_ASSERTE`, se parecen a esta macro pero emiten diferentes mensajes de diagnóstico definidos por el sistema o definidos por el usuario.

Consulte también

[#error \(directiva\) \(C/C++\)](#)
[assert \(macro\), _assert, _wassert](#)
[_ASSERT, _ASSERTE, _ASSERT_EXPR \(macros\)](#)
[static_assert](#)
[_STATIC_ASSERT \(Macro\)](#)
[Templates \(Plantillas \[C++\]\)](#)

static_assert

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

Comprueba una asercción de software en tiempo de compilación. Si la expresión constante especificada es `false`, el compilador muestra el mensaje especificado, si se produce uno, y se produce un error C2338 en la compilación; de lo contrario, la declaración no tiene ningún efecto.

Sintaxis

```
static_assert( constant-expression, string-literal );  
  
static_assert( constant-expression ); // C++17 (Visual Studio 2017 and  
later)
```

Parámetros

constant-expression

Una expresión constante entera que se puede convertir en un valor booleano. Si la expresión evaluada es cero (`false`), aparecerá el parámetro *string-literal* y se producirá un error de compilación. Si la expresión es distinta de cero (`true`), la declaración `static_assert` no tiene ningún efecto.

string-literal

Un mensaje que se muestra si el parámetro *constant-expression* es cero. El mensaje es una cadena de caracteres del [juego de caracteres base](#) del compilador; es decir, no de [caracteres anchos o multibyte](#).

Comentarios

El parámetro *constant-expression* de una declaración `static_assert` representa una *aserción de software*. Una asercción de software especifica una condición que se espera que sea cierta (valor `true`) en un determinado punto del programa. Si la condición es `true`, la declaración `static_assert` no tiene ningún efecto. Si la condición es `false`, se produce un error en la asercción, el compilador muestra el mensaje en el parámetro *string-literal* y se produce un error de compilación. En Visual Studio 2017 y versiones posteriores, el parámetro *string-literal* es opcional.

La declaración `static_assert` comprueba una aserción de software en tiempo de compilación. Por el contrario, las [funciones assert Macro y _assert y _wassert](#) prueban una aserción de software en tiempo de ejecución e incurren en un costo en tiempo de ejecución. La declaración `static_assert` resulta especialmente útil para depurar plantillas, porque los argumentos de plantilla se pueden incluir en el parámetro *constant-expression*.

El compilador examina si hay errores de sintaxis en la declaración `static_assert` al encontrar dicha declaración. El compilador evalúa el parámetro *constant-expression* inmediatamente si este no depende de un parámetro de plantilla. De lo contrario, el compilador evalúa el parámetro *constant-expression* cuando se crea una instancia de la plantilla. Por consiguiente, el compilador puede emitir un mensaje de diagnóstico una vez cuando se encuentra la declaración y otra vez cuando se crea una instancia de la plantilla.

Puede usar la palabra clave `static_assert` en el ámbito de espacio de nombres, clase o ámbito de bloque. (La palabra clave `static_assert` es técnicamente una declaración, aunque no introduce un nuevo nombre en el programa, porque se puede utilizar en el ámbito de espacio de nombres).

Descripción de `static_assert` con ámbito de espacio de nombres

En el ejemplo siguiente, la declaración `static_assert` tiene ámbito de espacio de nombres. Dado que el compilador conoce el tamaño del tipo `void *`, la expresión se evalúa inmediatamente.

Ejemplo: `static_assert` con ámbito de espacio de nombres

C++

```
static_assert(sizeof(void *) == 4, "64-bit code generation is not supported.");
```

Descripción de `static_assert` con ámbito de clase

En el ejemplo siguiente, la declaración `static_assert` tiene ámbito de clase.

`static_assert` comprueba si un parámetro de plantilla es de un tipo *Plain Old Data* (POD). El compilador examina la declaración `static_assert` cuando se declara, pero no evalúa el parámetro *constant-expression* hasta que se crean instancias de la plantilla de clase `basic_string` en `main()`.

Ejemplo: `static_assert` con ámbito de clase

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;
```

Descripción de `static_assert` con ámbito de bloque

En el ejemplo siguiente, la declaración `static_assert` tiene ámbito de bloque.

`static_assert` comprueba que el tamaño de la estructura de VMPage sea igual al valor `pagesize` de la memoria virtual del sistema.

Ejemplo: `static_assert` en el ámbito de bloque

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.");
        // ...
    }
// ...
};
```

Consulte también

[Aserción y mensajes proporcionados por el usuario \(C++\)](#)

[#error \(directiva\) \(C/C++\)](#)

[assert \(macro\), _assert, _wassert](#)

[Templates \(Plantillas \[C++\]\)](#)

[Juego de caracteres ASCII](#)

[Declaraciones y definiciones](#)

Información general de los módulos en C++

Artículo • 03/03/2023 • Tiempo de lectura: 9 minutos

C++20 presenta a los *módulos* con una solución moderna que convierte las bibliotecas y programas de C++ en componentes. Un *módulo* es un conjunto de archivos de código fuente que se compilan independientemente de las [unidades de traducción](#) que las importan. Los módulos eliminan o reducen muchos de los problemas asociados al uso de archivos de encabezado. A menudo reducen los tiempos de compilación. Las macros, las directivas de preprocesador y los nombres no exportados declarados en un módulo no son visibles fuera del módulo. No tienen efecto en la compilación de la unidad de traducción que importa el módulo. Puede importar módulos en cualquier orden sin preocuparse por las redefiniciones de las macros. Las declaraciones de la unidad de traducción de importación no participan en la resolución de sobrecarga ni en la búsqueda de nombres en el módulo importado. Una vez compilado un módulo, los resultados se almacenan en un archivo binario que describe todos los tipos, funciones y plantillas exportados. El compilador puede procesar ese archivo mucho más rápido que un archivo de encabezado. Además, el compilador puede reutilizarlo en cualquier lugar donde se importe el módulo en un proyecto.

Se pueden usar módulos en paralelo con los archivos de encabezado. Un archivo de código fuente de C++ puede importar `import` módulos y también incluir `#include` archivos de encabezado. En algunos casos, se puede importar un archivo de encabezado como un módulo en lugar de incluirlo textualmente mediante `#include` en el preprocesador. Se recomienda usar módulos en proyectos nuevos en lugar de archivos de encabezado tanto como sea posible. En el caso de los proyectos existentes de mayor envergadura y en desarrollo activo, experimente convertir encabezados heredados en módulos. Base su adopción sobre si obtiene una reducción significativa en los tiempos de compilación.

Para contrastar los módulos con otras formas de importar la biblioteca estándar, consulte [Comparar unidades de encabezado, módulos y encabezados precompilados](#).

Habilite los módulos en el compilador de Microsoft C++

A partir de la versión 17.1 de Visual Studio 2022, los módulos estándar de C++20 se implementan completamente en el compilador de Microsoft C++.

Antes de especificarlo el estándar de C++20, Microsoft tenía compatibilidad experimental con los módulos del compilador de C++ de Microsoft. El compilador también admite la importación de la biblioteca estándar como módulos, que se describe a continuación.

A partir de la versión 17.5 de Visual Studio 2022, la importación de la biblioteca estándar como módulo está estandarizada e implementada completamente en el compilador de C++ de Microsoft. En esta sección se describe el método experimental anterior, que todavía se admite. Para obtener información sobre la nueva forma estandarizada de importar la biblioteca estándar mediante módulos, consulte [Importación de la biblioteca estándar de C++ mediante módulos](#).

Se puede usar la característica de módulos para crear módulos de partición única e importar los módulos de la Biblioteca Estándar proporcionados por Microsoft. Para habilitar la compatibilidad con los módulos de la Biblioteca Estándar, compile con `/experimental:module` y `/std:c++latest`. En un proyecto de Visual Studio, haga clic con el botón derecho en el nodo del proyecto en el **Explorador de soluciones** y elija **Propiedades**. Establezca la lista desplegable **Configuración** en **Todas las configuraciones** y después elija **Propiedades de configuración C>/C++>Lenguaje>Habilitar módulos C++ (experimental)**.

Un módulo y el código que lo consume deben compilarse con las mismas opciones del compilador.

Consumo de la biblioteca estándar de C++ como módulos (experimental)

En esta sección se describe la implementación experimental, que todavía se admite. La nueva forma estandarizada de consumir la biblioteca estándar de C++ como módulos se describe en [Importación de la biblioteca estándar de C++ mediante módulos](#).

Al importar la Biblioteca Estándar de C++ como módulos en lugar de incluirla a través de archivos de encabezado, se pueden acelerar los tiempos de compilación en función del tamaño del proyecto. La biblioteca experimental se divide en los siguientes módulos con nombre:

- `std.regex` proporciona el contenido del encabezado `<regex>`
- `std.filesystem` proporciona el contenido del encabezado `<filesystem>`
- `std.memory` proporciona el contenido del encabezado `<memory>`
- `std.threading` proporciona el contenido de los encabezados `<atomic>`,
`<condition_variable>`, `<future>`, `<mutex>`, `<shared_mutex>` y `<thread>`

- `std.core` proporciona todo lo demás en la Biblioteca Estándar de C++

Para consumir estos módulos, agregue una declaración de importación a la parte superior del archivo de código fuente. Por ejemplo:

C++

```
import std.core;
import std.regex;
```

Para consumir los módulos de la Biblioteca Estándar de Microsoft, compile el programa con las opciones [/EHsc](#) y [/MD](#).

Ejemplo básico

En el ejemplo siguiente se muestra una definición de módulo simple en un archivo de código fuente denominado `Example.ixx`. La extensión `.ixx` es necesaria para los archivos de interfaz de módulo en Visual Studio. En este ejemplo, el archivo de interfaz contiene tanto la definición de función y como la declaración. Sin embargo, también se pueden colocar las definiciones en uno o varios archivos de implementación de módulo independientes, como se muestra en un ejemplo posterior. La instrucción `export module Example;` indica que este archivo es la interfaz principal de un módulo denominado `Example`. El modificador `export` en `f()` indica que esta función es visible cuando `Example` se importa en otro programa o módulo. El módulo hace referencia a un espacio de nombres `Example_NS`.

C++

```
// Example.ixx
export module Example;

#define ANSWER 42

namespace Example_NS
{
    int f_internal() {
        return ANSWER;
    }

    export int f() {
        return f_internal();
    }
}
```

El archivo `MyProgram.cpp` usa la declaración `import` para tener acceso al nombre exportado por `Example`. El nombre `Example_NS` está visible aquí, pero no todos sus miembros. Además, la macro `ANSWER` no está visible.

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
}
```

La declaración `import` solo puede aparecer en el ámbito global.

Gramática del módulo

`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 ;`

Implementación de módulos

Una *interfaz de módulo* exporta el nombre del módulo y todos los espacios de nombres, tipos, funciones, etc. que componen la interfaz pública del módulo. Una *implementación de módulo* define las cosas exportadas por el módulo. En su forma más sencilla, un módulo puede constar de un único archivo que combina la interfaz del módulo y la implementación. También se pueden colocar las implementaciones en uno o varios archivos de implementación de módulo independientes, de forma similar a cómo los archivos `.h` y `.cpp` se usan.

En el caso de los módulos más grandes, se pueden dividir partes del módulo en submódulos denominados *particiones*. Cada partición consta de un archivo de interfaz de módulo que exporta un nombre de partición de módulo. Una partición también puede tener uno o varios archivos de implementación de partición. El módulo en su conjunto tiene una *interfaz de módulo principal*, la interfaz pública del módulo que también puede importar y exportar las interfaces de partición.

Un módulo consta de una o más *unidades de módulo*. Una unidad de módulo es una unidad de traducción (un archivo de origen) que contiene una declaración de módulo. Hay varios tipos de unidades de módulo:

- Una *unidad de interfaz de módulo* es una unidad de módulo que exporta un nombre de módulo o un nombre de partición de módulo. Una unidad de interfaz de módulo tiene `export module` en su declaración de módulo.
- Una *unidad de implementación de módulo* es una unidad de módulo que no exporta un nombre de módulo o un nombre de partición de módulo. Como su nombre indica, se usa para implementar un módulo.
- Una *unidad de interfaz de módulo principal* es una unidad de interfaz de módulo que exporta el nombre del módulo. Debe haber una sola unidad de interfaz de módulo principal en un módulo.
- Una *unidad de interfaz de módulo* es una unidad de módulo de interfaz que exporta un nombre de partición de módulo.
- Una *unidad de implementación de particiones de módulo* es una unidad de implementación de módulo que tiene un nombre de partición de módulo en su declaración de módulo, pero no hay ninguna palabra clave `export`.

La palabra clave `export` solo se usa en archivos de interfaz. Un archivo de implementación puede importar `import` otro módulo, pero no puede exportar `export` ningún nombre. Los archivos de implementación pueden tener cualquier extensión.

Módulos, espacios de nombres y búsqueda dependiente de argumentos

Las reglas para los espacios de nombres en los módulos son las mismas que en cualquier otro código. Si se exporta una declaración dentro de un espacio de nombres, el espacio de nombres envolvente (excepto los miembros no exportados) también se exporta implícitamente. Si se exporta explícitamente un espacio de nombres, se exportan todas las declaraciones dentro de esa definición de espacio de nombres.

Cuando realiza una búsqueda dependiente del argumento para las resoluciones de sobrecarga en la unidad de traducción de importación, el compilador considera las funciones declaradas en la misma unidad de traducción (incluidas las interfaces del módulo) igual a donde se define el tipo de argumentos de la función.

Particiones de módulo

Una partición de módulo es similar a un módulo, excepto que comparte la propiedad de todas las declaraciones en todo el módulo. Todos los nombres exportados por los archivos de interfaz de la partición son importados y reexportados por el archivo de interfaz principal. El nombre de una partición debe comenzar con el nombre del módulo seguido de dos puntos. Las declaraciones de cualquiera de las particiones son visibles en todo el módulo. No se necesitan precauciones especiales para evitar los errores de la regla de definición única (ODR). Puede declarar un nombre (función, clase, etc.) en una partición y definirlo en otro. Un archivo de implementación de partición comienza de la siguiente manera:

C++

```
module Example:part1;
```

El archivo de interfaz de partición comienza de la siguiente manera:

C++

```
export module Example:part1;
```

Para acceder a las declaraciones de otra partición, una partición debe importarla, pero solo puede usar el nombre de la partición, no el nombre del módulo:

C++

```
module Example:part2;
import :part1;
```

La unidad de interfaz principal debe importar y volver a exportar todos los archivos de partición de la interfaz del módulo de la siguiente manera:

C++

```
export import :part1;
export import :part2;
...
```

La unidad de interfaz principal puede importar archivos de implementación de particiones, pero no puede exportarlos. Esos archivos no pueden exportar ningún nombre. Esta restricción permite a un módulo mantener los detalles de implementación internos en el módulo.

Módulos y archivos de encabezado

Se pueden incluir archivos de encabezado en un archivo de origen del módulo colocando la directiva `#include` antes de la declaración del módulo. Estos archivos se consideran en el *fragmento de módulo global*. Un módulo solo puede ver los nombres del fragmento de módulo global que se encuentran en los encabezados que incluye explícitamente. El fragmento de módulo global solo contiene símbolos que se usan.

C++

```
// MyModuleA.cpp

#include "customlib.h"
#include "anotherlib.h"

import std.core;
import MyModuleB;

//... rest of file
```

Se puede usar un archivo de encabezado tradicional para controlar qué módulos se importan:

C++

```
// MyProgram.h
import std.core;
```

```
#ifdef DEBUG_LOGGING
import std.filesystem;
#endif
```

Archivos de encabezado importados

Algunos encabezados son suficientemente independientes que se pueden incorporar mediante la palabra clave `import`. La principal diferencia entre un encabezado importado y un módulo importado es que las definiciones del preprocesador del encabezado son visibles en el programa de importación inmediatamente después de la instrucción `import`.

C++

```
import <vector>;
import "myheader.h";
```

Vea también

[module, import, export](#)

[Tutorial de módulos con nombre](#)

[Comparar unidades de encabezado, módulos y encabezados precompilados](#)

module, import, export

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Las declaraciones `module`, `import` y `export` están disponibles en C++20 y requieren el modificador del compilador `/experimental:module` junto con `/std:c++20` o posterior (como `/std:c++latest`). Para obtener más información, consulte [Información general de los módulos en C++](#).

module

Coloque una declaración `module` al principio de un archivo de implementación de módulo para especificar que el contenido del archivo pertenece al módulo nombrado.

C++

```
module ModuleA;
```

export

Use una declaración `export module` para el archivo de interfaz principal del módulo, que debe tener la extensión `.ixx`:

C++

```
export module ModuleA;
```

En un archivo de interfaz, use el modificador `export` en los nombres destinados a formar parte de la interfaz pública:

C++

```
// ModuleA.ixx

export module ModuleA;

namespace ModuleA_NS
{
    export int f();
    export double d();
    double internal_f(); // not exported
}
```

Los nombres no exportados no son visibles para el código que importa el módulo:

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
}
```

Es posible que la palabra clave `export` no aparezca en un archivo de implementación de módulo. Cuando `export` se aplica a un nombre de espacio de nombres, se exportan todos los nombres del espacio de nombres.

import

Use una declaración `import` para hacer que los nombres de un módulo sean visibles en el programa. La declaración `import` debe aparecer después de la declaración `module` y después de cualquier directiva `#include`, pero antes de cualquier declaración en el archivo.

C++

```
module ModuleA;

#include "custom-lib.h"
import std.core;
import std.regex;
import ModuleB;

// begin declarations here:
template <class T>
class Baz
{...};
```

Comentarios

Tanto `import` como `module` se tratan como palabras clave solo cuando aparecen al principio de una línea lógica:

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 ;
```

Específicos de Microsoft

En Microsoft C++, los tokens `import` y `module` son siempre identificadores y nunca palabras clave cuando se usan como argumentos para una macro.

Ejemplo

C++

```
#define foo(...) __VA_ARGS__  
foo(  
import // Always an identifier, never a keyword  
)
```

Fin de Específicos de Microsoft

Consulte también

[Información general de los módulos en C++](#)

Tutorial: Importación de la biblioteca estándar de C++ mediante módulos desde la línea de comandos

Artículo • 28/02/2023 • Tiempo de lectura: 15 minutos

Obtenga información sobre cómo importar la biblioteca estándar de C++ mediante módulos de biblioteca de C++. Esto es significativamente más rápido para compilar y más sólido que usar archivos de encabezado o unidades de encabezado o encabezados precompilados (PCH).

En este tutorial, obtenga información sobre:

- Cómo importar la biblioteca estándar como un módulo desde la línea de comandos.
- Ventajas de rendimiento y facilidad de uso de los módulos.
- Los dos módulos `std` de biblioteca estándar y `std.compat` la diferencia entre ellos.

Prerrequisitos

Este tutorial requiere Visual Studio 2022 17.5 o posterior.

Advertencia

Este tutorial es para una característica en versión preliminar. La característica está sujeta a cambios entre versiones preliminares. No debe usar características en versión preliminar en el código de producción.

Introducción a los módulos de la biblioteca estándar

Los archivos de encabezado sufren de semántica que cambian en función de las definiciones de macros, la semántica que cambia según el orden en que los incluya y ralentizan la compilación. Los módulos resuelven estos problemas.

Ahora es posible importar la biblioteca estándar como módulo en lugar de como una tanga de archivos de encabezado. Esto es significativamente más rápido y más sólido

que incluir archivos de encabezado o unidades de encabezado o encabezados precompilados (PCH).

La biblioteca estándar de C++23 presenta dos módulos con nombre: `std` y `std.compat`.

- `std` exporta las declaraciones y los nombres definidos en el espacio `std` de nombres de la biblioteca estándar de C++, como `std::vector` y `std::sort`. También exporta el contenido de los encabezados de contenedor de C, como `<cstdio>` y `<cstdlib>`, que proporcionan funciones como `std::printf()`. Las funciones de C definidas en el *espacio de nombres global*, como `::printf()`, no se exportan. Esto mejora la situación en la que se incluye un encabezado contenedor de C como `<cstdio>` también se incluyen archivos de encabezado de C como `stdio.h`, que incluyeron las versiones del espacio de nombres global de C. Este ya no es un problema si importa `std`.
- `std.compat` exporta todo en `std` y agrega los homólogos del espacio de nombres global del entorno de ejecución de C, como `::printf`, `::fopen`, `::size_t`, `::strlen`, etc. Este módulo facilita el trabajo con un código base que hace referencia a muchas funciones o tipos en tiempo de ejecución de C en el espacio de nombres global.

El compilador importa toda la biblioteca estándar cuando se usa `import std;` o `import std.compat;` y lo hace más rápido que incorporar un único archivo de encabezado. Es decir, es más rápido incorporar toda la biblioteca estándar con `import std;` (o `import std.compat`) que para `#include <vector>`, por ejemplo.

Dado que los módulos con nombre no exponen macros, las macros como `assert`, `errno offsetof`, `, va_arg`, y otras no están disponibles al importar `std` o `std.compat`. Consulte [Consideraciones sobre módulos con nombre de biblioteca estándar](#) para obtener soluciones alternativas.

Un poco sobre los módulos de C++

Los archivos de encabezado son cómo se han compartido declaraciones y definiciones entre archivos de origen en C++. Antes de los módulos de biblioteca estándar, incluiría la parte de la biblioteca estándar que necesitaba con una directiva como `#include <vector>`. Los archivos de encabezado son frágiles y difíciles de componer, ya que su semántica puede cambiar en función del orden en que los incluya, o si determinadas macros son o no están definidas. También ralentizan la compilación porque se vuelven a procesar por cada archivo de origen que los incluye.

C++20 presenta una alternativa moderna denominada *módulos*. En C++23, pudimos capitalizar la compatibilidad con módulos para introducir módulos con nombre para la biblioteca estándar.

Al igual que los archivos de encabezado, los módulos permiten compartir declaraciones y definiciones entre archivos de código fuente. Pero, a diferencia de los archivos de encabezado, los módulos no son frágiles y son más fáciles de componer porque su semántica no cambia debido a definiciones de macros o al orden en el que se importan. El compilador puede procesar módulos significativamente más rápido de lo que puede procesar `#include` archivos y también usa menos memoria en tiempo de compilación. Los módulos con nombre no exponen definiciones de macro ni detalles de implementación privada.

Para obtener información detallada sobre los [módulos](#), consulte [Información general sobre los módulos en C++](#). En este artículo también se describe cómo consumir la biblioteca estándar de C++ como módulos, pero se usa una forma más antigua y experimental de hacerlo.

En este artículo se muestra la nueva y mejor manera de consumir la biblioteca estándar. Para obtener más información sobre las formas alternativas de consumir la biblioteca estándar, vea [Comparar unidades de encabezado, módulos y encabezados precompilados](#).

Importación de la biblioteca estándar con `std`

En los ejemplos siguientes se muestra cómo consumir la biblioteca estándar como módulo mediante el compilador de línea de comandos. La experiencia del IDE de Visual Studio para importar la biblioteca estándar como módulo se está implementando.

La instrucción `import std;` o `import std.compat;` importa la biblioteca estándar en la aplicación. Pero en primer lugar, debe compilar la biblioteca estándar denominada `modules`. En los pasos siguientes se muestra cómo hacerlo.

Ejemplo: `std`

1. Abra un símbolo del sistema de herramientas nativas x86 para VS: en el menú **Inicio de Windows**, escriba *x86 nativo* y el símbolo del sistema debería aparecer en la lista de aplicaciones. Asegúrese de que el mensaje es para la versión preliminar 17.5 o posterior de Visual Studio 2022. Obtendrá errores si usa la versión incorrecta del símbolo del sistema. Los ejemplos que se usan en este tutorial son para el shell de CMD. Si usa PowerShell, sustituya

```
"$Env:VCToolsInstallDir\modules\std.ixx" por  
"%VCToolsInstallDir\modules\std.ixx".
```

2. Cree un directorio, como `%USERPROFILE%\source\repos\STLModules`, y consítelo como el directorio actual. Si elige un directorio al que no tiene acceso de escritura, obtendrá errores como, por ejemplo, no poder abrir el archivo `std.ifc` de salida .

3. Compile el `std` módulo con nombre con el siguiente comando:

dos

```
cl /std:c++latest /EHsc /nologo /W4 /MTd /c  
"%VCToolsInstallDir%\modules\std.ixx"
```

Si recibe errores, asegúrese de que usa la versión correcta del símbolo del sistema. Si sigue teniendo problemas, envíe un error en [Visual Studio Developer Community](#).

Debe compilar el `std` módulo con nombre con la misma configuración del compilador que pretende usar con el código que lo importa. Si tiene una solución de varios proyectos, puede compilar la biblioteca estándar denominada `module` una vez y, a continuación, hacer referencia a ella desde todos los proyectos mediante la `/reference` opción del compilador.

Con el comando anterior del compilador, el compilador genera dos archivos:

- `std.ifc` es la representación binaria compilada de la interfaz de módulo con nombre que el compilador consulta para procesar la `import std;` instrucción. Se trata de un artefacto solo en tiempo de compilación. No se envía con la aplicación.
- `std.obj` contiene la implementación del módulo con nombre. Agregue `std.obj` a la línea de comandos al compilar la aplicación de ejemplo para vincular estáticamente la funcionalidad que usa desde la biblioteca estándar a la aplicación.

Los modificadores de línea de comandos de clave de este ejemplo son:

| Modificador | Significado |
|------------------------------|--|
| <code>/std:c++:latest</code> | Use la versión más reciente del estándar y la biblioteca del lenguaje C++. Aunque la compatibilidad con módulos está disponible en <code>/std:c++20</code> , necesita la biblioteca estándar más reciente para obtener compatibilidad con la biblioteca estándar denominada <code>modules</code> . |

| Modificador | Significado |
|-------------|--|
| /EHsc | Use el control de excepciones de C++, excepto para las funciones marcadas como <code>extern "C"</code> . |
| /MTd | Use la biblioteca en tiempo de ejecución de depuración multiproceso estática. |
| /c | Compile sin vinculación. |

4. Para probar la importación de la `std` biblioteca, empiece por crear un archivo denominado `importExample.cpp` con el siguiente contenido:

```
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;
    }
}
```

En el código anterior, `import std;` reemplaza `#include <vector>` y `#include <iostream>`. La instrucción `import std;` hace que toda la biblioteca estándar esté disponible con una instrucción . Si le preocupa el rendimiento, puede ayudar a saber que la importación de toda la biblioteca estándar suele ser significativamente más rápida que procesar un único archivo de encabezado de biblioteca estándar, como `#include <vector>`.

5. Compile el ejemplo mediante el siguiente comando en el mismo directorio que el paso anterior:

```
dos

cl /std:c++latest /EHsc /nologo /W4 /MTd importExample.cpp std.obj
```

No teníamos que especificar `std.ifc` en la línea de comandos porque el compilador busca automáticamente el `.ifc` archivo que coincide con el nombre del módulo especificado por una `import` instrucción . Cuando el compilador

encuentra `import std;`, encuentra, ya que `std.ifc` lo colocamos en el mismo directorio que el código fuente, lo que nos permite especificarlo en la línea de comandos. Si el `.ifc` archivo está en un directorio diferente al código fuente, use el [/reference](#) modificador del compilador para hacer referencia a él.

Si va a compilar un solo proyecto, puede combinar los pasos para compilar la `std` biblioteca estándar denominada module y el paso de compilar la aplicación agregando `"%VCToolsInstallDir%\modules\std.ixx"` a la línea de comandos.

Asegúrese de colocarlo antes de cualquier `.cpp` archivo que lo consuma. De forma predeterminada, el nombre del ejecutable de salida se toma del primer archivo de entrada. Use la `/Fe` opción del compilador para especificar el nombre de archivo de salida que desee. En este tutorial se muestra cómo compilar el `std` módulo con nombre como un paso independiente porque solo necesita compilar la biblioteca estándar denominada module una vez y, a continuación, puede hacer referencia a él desde el proyecto o desde varios proyectos. Pero puede ser conveniente compilar todo juntos, como se muestra en esta línea de comandos:

```
dos
```

```
cl /FeimportExample /std:c++latest /EHsc /nologo /W4 /MTd  
"%VCToolsInstallDir%\modules\std.ixx" importExample.cpp
```

Dada la línea de comandos anterior, el compilador genera un archivo ejecutable denominado `importExample.exe`. Al ejecutarlo, genera la siguiente salida:

```
Resultados
```

```
Import the STL library for best performance  
555
```

Los modificadores de línea de comandos de clave de este ejemplo son los mismos que el ejemplo anterior, salvo que no se usa el `/c` modificador.

Importación de la biblioteca estándar y las funciones globales de C con `std.compat`

La biblioteca estándar de C++ incluye la biblioteca estándar ISO C. El `std.compat` módulo proporciona toda la funcionalidad del `std` módulo, como `std::vector`, `std::cout`, `std::printf` `std::scanf`, etc. Pero también proporciona las versiones del

espacio de nombres global de estas funciones, como `::printf`, `::scanf`, `::fopen`, `::size_t`, etc.

El `std.compat` módulo con nombre es una capa de compatibilidad proporcionada para facilitar la migración del código existente que hace referencia a las funciones en tiempo de ejecución de C en el espacio de nombres global. Si desea evitar agregar nombres al espacio de nombres global, use `import std;`. Si necesita facilitar la migración de un código base que usa muchas funciones en tiempo de ejecución de C sin calificar (es decir, espacio de nombres global), use `import std.compat;`. Esto proporciona los nombres de tiempo de ejecución del espacio de nombres global de C para que no tenga que calificar todas las menciones de nombre global con `std::`. Si no tiene ningún código existente que use las funciones en tiempo de ejecución del espacio de nombres global de C, no es necesario usar `import std.compat;`. Si solo llama a algunas funciones en tiempo de ejecución de C en el código, puede ser mejor usar `import std;` y calificar los pocos nombres de tiempo de ejecución de espacio de nombres globales de C que lo necesitan con `std::`, por ejemplo, `std::printf()`. Si ve un error similar `error C3861: 'printf': identifier not found` al intentar compilar el código, considere la posibilidad de usar `import std.compat;` para importar las funciones en tiempo de ejecución del espacio de nombres global de C.

Ejemplo: `std.compat`

Para poder usar `import std.compat;` en el código, debe compilar el módulo `std.compat.ixx` con nombre . Los pasos son similares a para compilar el `std` módulo con nombre. Los pasos incluyen compilar primero el `std` módulo con nombre porque `std.compat` depende de él:

1. Abra un símbolo del sistema de Herramientas nativas para VS: en el menú **Inicio** de Windows, escriba *x86 nativo* y el símbolo del sistema debería aparecer en la lista de aplicaciones. Asegúrese de que el mensaje es para la versión preliminar 17.5 o posterior de Visual Studio 2022. Obtendrá errores del compilador si usa la versión incorrecta del símbolo del sistema.
2. Cree un directorio para probar este ejemplo, como `%USERPROFILE%\source\repos\STLModules` y convertirlo en el directorio actual. Si elige un directorio al que no tiene acceso de escritura, obtendrá errores como, por ejemplo, no poder abrir el archivo `std.ifc` de salida .
3. Compile los `std` módulos con nombre y `std.compat` con el siguiente comando:

```
cl /std:c++latest /EHsc /nologo /W4 /MTd /c  
"%VCToolsInstallDir%\modules\std.ixx"  
"%VCToolsInstallDir%\modules\std.compat.ixx"
```

Debe compilar `std` y `std.compat` usar la misma configuración del compilador que pretende usar con el código que los importa. Si tiene una solución de varios proyectos, puede compilarlas una vez y, a continuación, hacer referencia a ellas desde todos los proyectos mediante la [/reference](#) opción del compilador.

Si recibe errores, asegúrese de que usa la versión correcta del símbolo del sistema. Si sigue teniendo problemas, envíe un error en [Visual Studio Developer Community](#). Aunque esta característica todavía está en versión preliminar, puede encontrar una lista de problemas [conocidos en Unidades de encabezado de biblioteca estándar y módulos de seguimiento del problema 1694](#).

El compilador genera cuatro archivos de los dos pasos anteriores:

- `std.ifc` es la interfaz de módulo con nombre binario compilado que el compilador consulta para procesar la `import std;` instrucción. El compilador también consulta `std.ifc` para procesar `import std.compat;` porque `std.compat` se basa en `std`. Se trata de un artefacto solo en tiempo de compilación. No se envía con la aplicación.
- `std.obj` contiene la implementación de la biblioteca estándar.
- `std.compat.ifc` es la interfaz de módulo con nombre binario compilado que el compilador consulta para procesar la `import std.compat;` instrucción. Se trata de un artefacto solo en tiempo de compilación. No se envía con la aplicación.
- `std.compat.obj` contiene la implementación. Sin embargo, la mayoría de la implementación la proporciona `std.obj`. Agregue `std.obj` a la línea de comandos al compilar la aplicación de ejemplo para vincular estáticamente la funcionalidad que se usa desde la biblioteca estándar a la aplicación.

4. Para probar la importación de la `std.compat` biblioteca, cree un archivo denominado `stdCompatExample.cpp` con el siguiente contenido:

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);
    }
}
```

En el código anterior, `import std.compat;` reemplaza `#include <cstdio>` y `#include <vector>`. La instrucción `import std.compat;` hace que la biblioteca estándar y las funciones en tiempo de ejecución de C estén disponibles con una instrucción . Si le preocupa el rendimiento, el rendimiento de los módulos es tal que importar este módulo con nombre, que incluye la biblioteca estándar de C++ y las funciones de espacio de nombres globales de la biblioteca en tiempo de ejecución de C, es más rápido que incluso procesar un único incluido como `#include <vector>`.

5. Compile el ejemplo mediante el siguiente comando:

```
dos

cl /std:c++latest /EHsc /nologo /W4 /MTd stdCompatExample.cpp std.obj
std.compat.obj
```

No teníamos que especificar `std.compat.ifc` en la línea de comandos porque el compilador busca automáticamente el `.ifc` archivo que coincide con el nombre del módulo en una `import` instrucción . Cuando el compilador encuentra `import std.compat;` que se encuentra `std.compat.ifc` , ya que lo colocamos en el mismo directorio que el código fuente, lo que nos alivia de la necesidad de especificarlo en la línea de comandos. Si el `.ifc` archivo está en un directorio diferente al código fuente, use el `/reference` modificador del compilador para hacer referencia a él.

Aunque se va a importar `std.compat`, también debe vincularse a `std.obj` porque es donde reside la implementación de la biblioteca estándar que `std.compat` se basa.

Si va a compilar un solo proyecto, puede combinar el paso de compilación de la `std` biblioteca estándar denominada `std.compat` modules con el paso de compilar la aplicación agregando `"%VCToolsInstallDir%\modules\std.ixx"` y `"%VCToolsInstallDir%\modules\std.compat.ixx"` (en ese orden) a la línea de comandos. Asegúrese de colocarlos antes de los `.cpp` archivos que los consuman y especifique `/Fe` el nombre compilado `exe` como se muestra en este ejemplo:

```
dos
```

```
c1 /FestdCompatExample /std:c++latest /EHsc /nologo /W4 /MTd  
"%VCToolsInstallDir%\modules\std.ixx"  
"%VCToolsInstallDir%\modules\std.compat.ixx" stdCompatExample.cpp
```

Los he mostrado como pasos independientes en este tutorial porque solo necesita compilar la biblioteca estándar denominada modules una vez y, a continuación, puede hacer referencia a ellos desde el proyecto o desde varios proyectos. Pero puede ser conveniente construirlos a la vez.

El comando anterior del compilador genera un archivo ejecutable denominado `stdCompatExample.exe`. Al ejecutarlo, genera la siguiente salida:

```
Resultados
```

```
Import std.compat to get global names like printf()  
555
```

Consideraciones sobre módulos con nombre de biblioteca estándar

El control de versiones de los módulos con nombre es el mismo que para los encabezados. Los `.ixx` archivos de módulo con nombre residen junto con los encabezados, por ejemplo: `"%VCToolsInstallDir%\modules\std.ixx`, que se resuelve en la versión de las herramientas usadas `C:\Program Files\Microsoft Visual Studio\2022\Preview\VC\Tools\MSVC\14.35.32019\modules\std.ixx` en el momento de redactar este documento. Seleccione la versión del módulo con nombre para usar de la misma manera que elija la versión del archivo de encabezado desde la que va a usar el directorio desde el que hace referencia.

No mezcle ni coincida con la importación de unidades de encabezado y módulos con nombre. Por ejemplo, no `import <vector>;` y `import std;` en el mismo archivo.

No mezcle ni coincida con la importación de archivos de encabezado de biblioteca estándar de C++ y módulos con nombre. Por ejemplo, no `#include <vector>` y `import std;` en el mismo archivo. Sin embargo, puede incluir encabezados C e importar módulos con nombre en el mismo archivo. Por ejemplo, puede `import std;` y `#include <math.h>` en el mismo archivo. Simplemente no incluya la versión de la biblioteca estándar de C++, `<cmath>`.

No es necesario defenderse de importar un módulo varias veces: no se requiere protección de encabezados `#ifndef`. El compilador sabe cuándo ya ha importado un módulo con nombre y omite los intentos duplicados de hacerlo.

Si necesita usar la `assert()` macro, entonces `#include <assert.h>`.

Si necesita usar la `errno` macro, `#include <errno.h>`. Dado que los módulos con nombre no exponen macros, esta es la solución alternativa si necesita comprobar si hay errores de `<math.h>`, por ejemplo.

Las macros como `NAN`, `INFINITY`y `INT_MIN` se definen mediante `<limits.h>`, que se pueden incluir. Sin embargo, si `import std;` puede usar `numeric_limits<double>::quiet_NaN()` y `numeric_limits<double>::infinity()` en lugar de `NAN` y `INFINITY`, y `std::numeric_limits<int>::min()` en lugar de `INT_MIN`.

Resumen

En este tutorial, ha importado la biblioteca estándar mediante módulos. A continuación, obtenga información sobre cómo crear e importar sus propios módulos en el tutorial [Módulos con nombre en C++](#).

Consulta también

[Comparación de unidades de encabezado, módulos y encabezados precompilados](#)

[Información general de los módulos en C++](#)

[Un recorrido por los módulos de C++ en Visual Studio ↗](#)

[Traslado de un proyecto a módulos con nombre de C++ ↗](#)

Tutorial de módulos con nombre (C++)

Artículo • 15/02/2023 • Tiempo de lectura: 24 minutos

Este tutorial trata sobre la creación de módulos de C++20. Los módulos reemplazan los archivos de encabezado. Obtendrá información sobre cómo los módulos son una mejora respecto a los archivos de encabezado.

En este tutorial, aprenderá a:

- Creación e importación de un módulo
- Creación de una unidad de interfaz principal del módulo
- Creación de un archivo de partición de módulo
- Creación de un archivo de implementación de unidad de módulo

Requisitos previos

Este tutorial requiere Visual Studio 2022 17.1.0 o posterior.

Es posible que obtenga errores de IntelliSense mientras trabaja en el ejemplo de código de este tutorial. El trabajo en el motor de IntelliSense se está actualizando con el compilador. Los errores de IntelliSense se pueden omitir y no impedirán que se compile el ejemplo de código. Para realizar un seguimiento del progreso en el trabajo de IntelliSense, consulte este [problema](#).

Qué son los módulos de C++

Los archivos de encabezado indican cómo se comparten las declaraciones y las definiciones entre archivos de código fuente en C++. Los archivos de encabezado son frágiles y difíciles de componer. Pueden compilarse de forma diferente en función del orden en el que los incluya o en las macros que son o no están definidas. Pueden ralentizar el tiempo de compilación porque se vuelven a procesar para cada archivo de código fuente que los incluya.

C++20 presenta un enfoque moderno para la creación de componentes a partir de programas de C++: los *módulos*.

Al igual que los archivos de encabezado, los módulos permiten compartir declaraciones y definiciones entre archivos de código fuente. Pero a diferencia de los archivos de encabezado, los módulos no filtran definiciones de macros ni detalles privados de la implementación.

Los módulos son más fáciles de componer porque su semántica no cambia debido a definiciones de macros o a lo que se ha importado, el orden de las importaciones, etc. También facilitan el control de lo que es visible para los consumidores.

Los módulos proporcionan garantías de seguridad adicionales que los archivos de encabezado no proporcionan. El compilador y el enlazador trabajan juntos para evitar posibles problemas de colisión de nombres y proporcionan garantías más sólidas de una regla de definición ([ODR ↴](#)).

Un modelo de propiedad segura evita conflictos entre nombres en el momento del enlazado porque el enlazador adjunta los nombres exportados al módulo que los exporta. Este modelo permite que el compilador de Microsoft Visual C++ evite un comportamiento indefinido causado por el enlazado de distintos módulos que notifican nombres similares en el mismo programa. Para obtener más información, consulte [Propiedad segura ↴](#).

Un módulo se compone de uno o varios archivos de código fuente compilados en un archivo binario. El archivo binario describe todos los tipos, funciones y plantillas exportados del módulo. Cuando un archivo de código fuente importa un módulo, el compilador lee en el archivo binario que tiene el contenido del módulo. Leer el archivo binario es mucho más rápido que procesar un archivo de encabezado. Además, el compilador reutiliza el archivo binario cada vez que se importa el módulo, lo que ahorra aún más tiempo. Dado que un módulo se compila una vez en lugar de cada vez que se importa, se puede reducir el tiempo de compilación, a veces drásticamente.

Más importante, los módulos no tienen los problemas de fragilidad que hacen los archivos de encabezado. La importación de un módulo no cambia la semántica del módulo ni la semántica de ningún otro módulo importado. Las macros, las directivas de preprocesador y los nombres no exportados declarados en un módulo no son visibles para el archivo de código fuente que lo importa. Puede importar módulos en cualquier orden y no cambiará el significado de los módulos.

Los módulos se pueden usar en paralelo con los archivos de encabezado. Esta característica es conveniente si va a migrar una base de código para usar módulos porque puede hacerlo en fases.

En algunos casos, un archivo de encabezado se puede importar como una unidad de encabezado en lugar de como un archivo `#include`. Las unidades de encabezado son la alternativa recomendada a los [archivos de encabezado precompilado](#) (PCH). Son más fáciles de configurar y de usar que los archivos [PCH compartidos ↴](#), pero proporcionan ventajas de rendimiento similares. Para obtener más información, consulte [Tutorial: Compilación e importación de unidades de encabezado en Microsoft Visual C++](#).

El código puede consumir módulos en el mismo proyecto o cualquier proyecto al que se haga referencia automáticamente mediante referencias de proyecto a proyecto a proyectos de biblioteca estática.

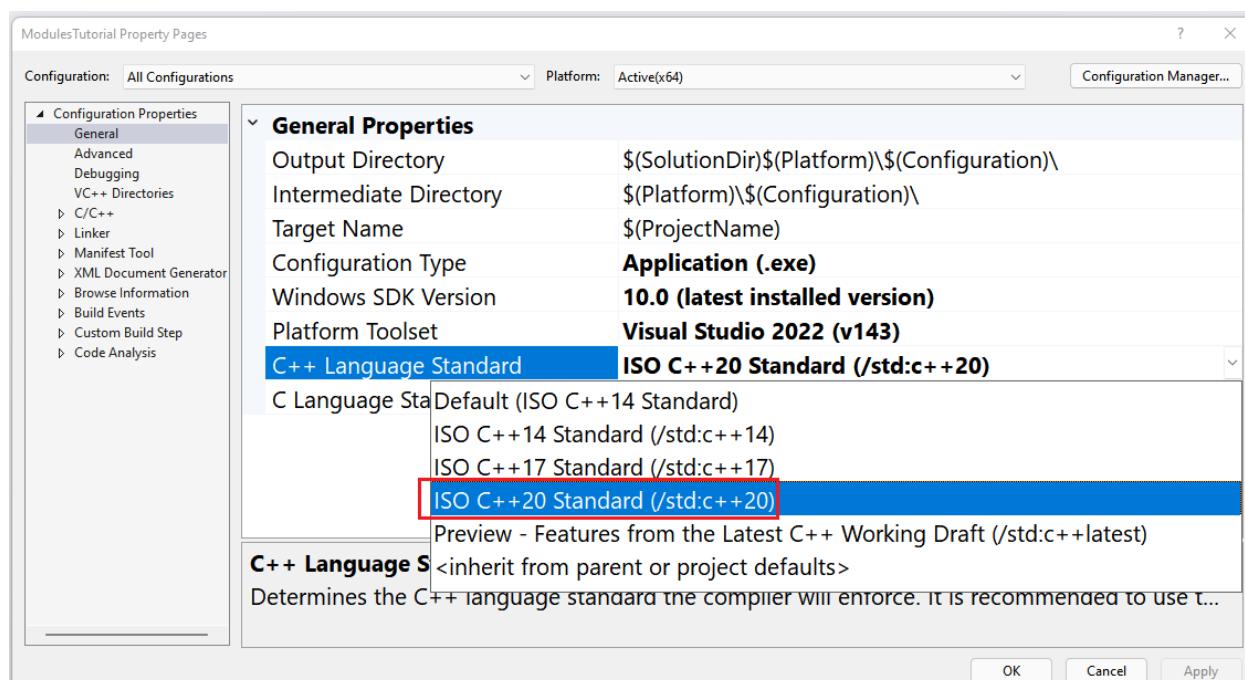
Creación del proyecto

A medida que compilamos un proyecto sencillo, veremos varios aspectos de los módulos. El proyecto implementará una API mediante un módulo en lugar de un archivo de encabezado.

En Visual Studio 2022 o posterior, elija **Crear un nuevo proyecto** y, a continuación, el tipo de proyecto **Aplicación de consola** (para C++). Si este tipo de proyecto no está disponible, es posible que no haya seleccionado la carga de trabajo **Desarrollo de escritorio con C++** al instalar Visual Studio. Puede usar el Instalador de Visual Studio para agregar la carga de trabajo de C++.

Asigne al nuevo proyecto el nombre `ModulesTutorial` y cree el proyecto.

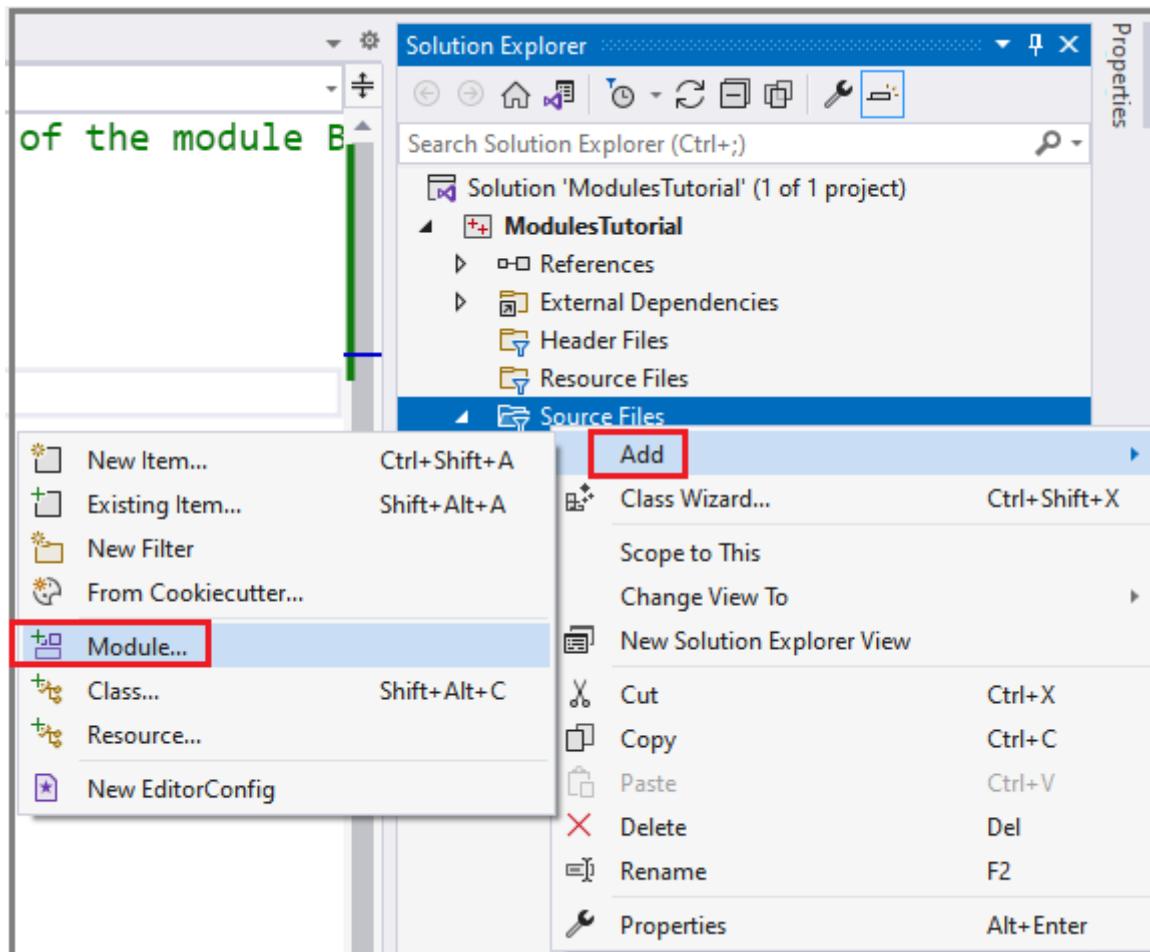
Dado que los módulos son una característica de C++20, use la [/std:c++20](#) opción del compilador o [/std:c++latest](#). En el **Explorador de soluciones**, haga clic con el botón derecho en el nombre `ModulesTutorial` del proyecto y, a continuación, elija **Propiedades**. En el cuadro de diálogo Páginas de propiedades del proyecto, cambie **Configuración** a **Todas las configuraciones** y **Plataforma** a **Todas las plataformas**. Seleccione **Propiedades de configuración > General** en el panel de vista de árbol de la izquierda. Seleccione la propiedad **Estándar del lenguaje C++**. Use la lista desplegable para cambiar el valor de la propiedad a **Estándar ISO C++20 (/std:c++20)**. Seleccione **Aceptar** para aceptar el cambio.



Creación de la unidad de interfaz principal del módulo

Un módulo consta de uno o varios archivos. Uno de estos archivos debe ser lo que se conoce como *unidad de interfaz principal del módulo*. Define lo que exporta el módulo; es decir, lo que verán los importadores del módulo. Solo puede haber una unidad de interfaz de módulo principal por módulo.

Para agregar una unidad de interfaz de módulo principal, en **Explorador de soluciones**, haga clic con el botón derecho en **Archivos de origen** y seleccione **Agregar>módulo**.



En el cuadro de diálogo **Agregar nuevo elemento** que aparece, asigne al nuevo módulo el nombre `BasicPlane.Figures.ixx` y elija **Agregar**.

El contenido predeterminado del archivo de módulo creado tiene dos líneas:

```
C++  
  
export module BasicPlane;  
  
export void MyFunc();
```

Las `export module` palabras clave de la primera línea declaran que este archivo es una unidad de interfaz de módulo. Aquí hay un punto sutil: para cada módulo con nombre, debe haber exactamente una unidad de interfaz de módulo sin ninguna partición de módulo especificada. Esa unidad del módulo se llama *unidad de interfaz principal del módulo*.

La unidad de interfaz principal del módulo es donde se declaran las funciones, los tipos, las plantillas, otros módulos y las particiones de módulo que se van a exponer cuando los archivos de código fuente importan el módulo. Un módulo puede constar de varios archivos, pero solo el archivo de interfaz principal del módulo identifica lo que se va a exponer.

Reemplace el contenido del `BasicPlane.Figures.ixx` archivo por:

C++

```
export module BasicPlane.Figures; // the export module keywords mark this
file as a primary module interface unit
```

Esta línea identifica este archivo como la interfaz principal del módulo y proporciona al módulo un nombre: `BasicPlane.Figures`. El punto del nombre del módulo no tiene ningún significado especial para el compilador. Se puede usar un punto para transmitir cómo está organizado el módulo. Si tiene varios archivos de módulo que funcionan juntos, puede usar puntos para indicar una separación de intereses. En este tutorial, usaremos puntos para indicar las diferentes áreas funcionales de la API.

Este nombre también es donde procede el "con nombre" de "módulo con nombre". Los archivos que forman parte de este módulo usan este nombre para identificarse como parte del módulo con nombre. Un módulo con nombre es la colección de unidades de módulo con el mismo nombre de módulo.

Deberíamos hablar un momento sobre la API que vamos a implementar antes de continuar. Afecta a las decisiones que hacemos a continuación. La API representa diferentes formas. Solo vamos a proporcionar un par de formas en este ejemplo: `Point` y `Rectangle`. `Point` está pensado para usarse como parte de formas más complejas, como `Rectangle`.

Para ilustrar algunas características de los módulos, vamos a factorizar esta API en partes. Una parte será la `Point` API. La otra parte será la relativa a `Rectangle`. Imagine que esta API va a crecer para convertirse en algo más complejo. La división es útil para separar problemas o facilitar el mantenimiento del código.

Hasta ahora, hemos creado la interfaz principal del módulo que expondrá esta API. Ahora, vamos a crear la API `Point`. Queremos que forme parte de este módulo. Por motivos de organización lógica y posible eficacia de la compilación, queremos que esta parte de la API sea fácil de entender por sí misma. Para ello, crearemos un archivo de *partición de módulo*.

Un archivo de partición de módulo es una parte, o un componente, de un módulo. Lo que hace que sea único es que se puede tratar como una parte individual del módulo, pero solo dentro del módulo. Las particiones de módulo no se pueden consumir fuera de un módulo. Las particiones de módulo son útiles para dividir la implementación del módulo en partes administrables.

Al importar una partición en el módulo principal, todas sus declaraciones serán visibles para el módulo principal, independientemente de si se exportan. Las particiones se pueden importar en cualquier interfaz de partición, interfaz principal del módulo o unidad de módulo que pertenezca al módulo con nombre.

Creación de un archivo de partición de módulo

Partición de módulo `Point`

Para crear un archivo de partición de módulo, en el **Explorador de soluciones** haga clic con el botón derecho en **Archivos de origen** y seleccione **Agregar>módulo**. Asigne el nombre al archivo `BasicPlane.Figures-Point.ixx` y seleccione **Agregar**.

Dado que es un archivo de partición de módulo, hemos agregado un guión y el nombre de la partición al nombre del módulo. Esta convención ayuda al compilador en el caso de la línea de comandos porque el compilador usa reglas de búsqueda de nombres basadas en el nombre del módulo para buscar el archivo compilado `.ifc` para la partición. De este modo, no es necesario proporcionar argumentos de la línea de comandos de tipo `/reference` explícitos para buscar las particiones que pertenecen al módulo. También resulta útil organizar los archivos que pertenecen a un módulo por su nombre, ya que puede ver fácilmente qué archivos pertenecen a los módulos.

Reemplace el contenido de `BasicPlane.Figures-Point.ixx` por:

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;  
};
```

El archivo empieza con `export module`. Estas palabras clave también son cómo comienza la interfaz del módulo principal. Lo que hace que este archivo sea diferente es el signo de dos puntos (`:`) que sigue al nombre del módulo, seguido del nombre de la partición. Esta convención de nomenclatura identifica el archivo como una *partición de módulo*. Dado que define la interfaz del módulo para una partición, no se considera la interfaz principal del módulo.

El nombre `BasicPlane.Figures:Point` identifica esta partición como parte del módulo `BasicPlane.Figures`. (Recuerde que el punto del nombre no tiene ningún significado especial para el compilador). Los dos puntos indican que este archivo contiene una partición de módulo denominada `Point` que pertenece al módulo `BasicPlane.Figures`. Podemos importar esta partición en otros archivos que formen parte de este módulo con nombre.

En este archivo, la palabra clave `export` hace que `struct Point` sea visible para los consumidores.

Partición de módulo `Rectangle`

La siguiente partición que definiremos es `Rectangle`. Cree otro archivo de módulo con los mismos pasos que antes: en **Explorador de soluciones**, haga clic con el botón derecho en **Archivos** de origen y seleccione **Agregar>módulo**. Asigne el nombre `BasicPlane.Figures-Rectangle.ixx` al archivo y seleccione **Agregar**.

Reemplace el contenido de `BasicPlane.Figures-Rectangle.ixx` por:

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);
```

El archivo comienza con `export module BasicPlane.Figures:Rectangle;` el que declara una partición de módulo que forma parte del módulo `BasicPlane.Figures`. El elemento `:Rectangle` agregado al nombre del módulo lo define como una partición del módulo `BasicPlane.Figures`. Se puede importar individualmente en cualquiera de los archivos de módulo que forman parte de este módulo con nombre.

A continuación, `import :Point;` muestra cómo importar una partición de módulo. La instrucción `import` hace que todos los tipos, funciones y plantillas exportados de la partición de módulo sean visibles para el módulo. No es necesario especificar el nombre del módulo. El compilador sabe que este archivo pertenece al módulo `BasicPlane.Figures` debido al elemento `export module BasicPlane.Figures:Rectangle;` de la parte superior del archivo.

A continuación, el código exporta la definición de `struct Rectangle` y las declaraciones de algunas funciones que devuelven varias propiedades del rectángulo. La palabra clave `export` indica si se debe hacer visible aquello a lo que precede para los consumidores del módulo. Se usa para hacer que las funciones `area`, `height` y `width` sean visibles fuera del módulo.

Todas las definiciones y declaraciones de una partición de módulo son visibles para la unidad de módulo de importación, tanto si tienen la `export` palabra clave como si no. La palabra clave `export` rige si la definición, la declaración o la definición de tipo estarán visibles fuera del módulo al exportar la partición en la interfaz principal del módulo.

Los nombres se hacen visibles para los consumidores de un módulo de varias maneras:

- Coloque la palabra clave `export` delante de cada tipo, función, etc., que quiera exportar.
- Si coloca `export` delante de un espacio de nombres, por ejemplo `export namespace N { ... }`, se exporta todo lo definido dentro de las llaves. Sin embargo, si en otra parte del módulo define `namespace N { struct S {...}; }`, `struct S` no está disponible para los consumidores del módulo. No está disponible porque esa declaración de espacio de nombres no está precedida por `export`, aunque haya otro espacio de nombres con el mismo nombre.
- Si no se debe exportar un tipo, una función, etc., omita la palabra clave `export`. Será visible para otros archivos que forman parte del módulo, pero no para los importadores del módulo.

- Use `module :private;` para marcar el principio de la partición privada de módulo.
La partición privada de módulo es una sección del módulo donde las declaraciones solo son visibles para ese archivo. No son visibles para los archivos que importan este módulo ni para otros archivos que forman parte de este módulo. Piense en ella como una sección que es estática local para el archivo. Esta sección solo está visible dentro del archivo.
- Para que un módulo o una partición de módulo importados sean visibles, use `export import`. En la sección siguiente, se muestra un ejemplo.

Redacción de las particiones de módulo

Ahora que tenemos las dos partes de la API definidas, vamos a reunirlas para que los archivos que importen este módulo puedan acceder a ellas en su conjunto.

Todas las particiones de módulo se deben exponer como parte de la definición del módulo al que pertenecen. Las particiones se exponen en la interfaz principal del módulo. Abra el archivo `BasicPlane.Figures.ixx`, que define la interfaz principal del módulo. Reemplace su contenido por lo siguiente:

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
```

Las dos líneas que comienzan por `export import` son nuevas aquí. Cuando se combina de esta forma, estas dos palabras clave indican al compilador que importe el módulo especificado y haga que sea visible para los consumidores de este módulo. En este caso, los dos puntos (`:`) del nombre del módulo indican que estamos importando una partición de módulo.

Los nombres importados no incluyen el nombre completo del módulo. Por ejemplo, la partición `:Point` se declaró como `export module BasicPlane.Figures:Point`. Sin embargo, aquí vamos a importar `:Point`. Dado que estamos en el archivo de interfaz del módulo principal para el módulo `BasicPlane.Figures`, el nombre del módulo está implícito y solo se especifica el nombre de la partición.

Hasta ahora, hemos definido la interfaz principal del módulo, que expone la superficie de API que queremos que esté disponible. Pero solo hemos declarado, no definido,

`area()`, `height()` y `width()`. Lo haremos a continuación mediante la creación de un archivo de implementación de módulo.

Creación de un archivo de implementación de unidad de módulo

Los archivos de implementación de unidad de módulo no terminan con una `.ixx` extensión, ya que son archivos normales `.cpp`. Para agregar un archivo de implementación de unidad de módulo, cree un archivo de código fuente mediante un clic con el botón derecho en el **Explorador de soluciones** en **Archivos de código fuente**, seleccione **Agregar>Nuevo elemento** y, a continuación, seleccione **Archivo de C++ (.cpp)**. Asigne al nuevo archivo el nombre `BasicPlane.Figures-Rectangle.cpp` y elija **Agregar**.

La convención de nomenclatura del archivo de implementación de la partición del módulo sigue la convención de nomenclatura de una partición. Pero tiene la extensión `.cpp` porque es un archivo de implementación.

Reemplace el contenido del archivo `BasicPlane.Figures-Rectangle.cpp` por lo siguiente:

```
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; }
```

Este archivo comienza con `module;` el que se introduce un área especial del módulo denominado *fragmento de módulo global*. Precede al código del módulo con nombre y es donde puede usar directivas de preprocesador como `#include`. El código del fragmento de módulo global no es propiedad de la interfaz del módulo ni lo exporta.

Al incluir un archivo de encabezado, normalmente no querrá que se trate como una parte exportada del módulo. Normalmente, el archivo de encabezado se incluye como un detalle de implementación que no debe formar parte de la interfaz del módulo. Puede haber casos avanzados en los que quiera hacerlo, pero generalmente no será así. No se generan metadatos independientes (archivos `.ifc`) para las directivas `#include`

del fragmento global del módulo. Los fragmentos globales del módulo proporcionan un buen lugar para incluir archivos de encabezado como `windows.h` o, en Linux, `unistd.h`.

El archivo de implementación del módulo que estamos creando no incluye ninguna biblioteca porque no las necesita como parte de su implementación. Pero si las necesita, esta área es donde irían las directivas `#include`.

La línea `module BasicPlane.Figures:Rectangle;` indica que este archivo forma parte del módulo `BasicPlane.Figures` con nombre `.`. El compilador trae automáticamente a este archivo las funciones y los tipos expuestos por la interfaz principal del módulo. Una unidad de implementación de módulo no tiene la `export` palabra clave antes de la `module` palabra clave en su declaración de módulo.

A continuación, está la definición de las funciones `area()`, `height()` y `width()`. Se declararon en la partición `Rectangle` en `BasicPlane.Figures-Rectangle.ixx`. Dado que la interfaz principal de este módulo importó las particiones de módulo `Point` y `Rectangle`, esos tipos están visibles aquí en el archivo de implementación de unidad de módulo. Una característica interesante de las unidades de implementación de módulo: el compilador hace que todo lo que hay en la interfaz principal del módulo correspondiente sea visible para el archivo. No es necesario utilizar `imports <module-name>`.

Todo lo que declare dentro de una unidad de implementación solo es visible para el módulo al que pertenece.

Importación del módulo

Ahora, vamos a usar el módulo que hemos definido. Abra el archivo `ModulesTutorial.cpp`. Se creó automáticamente como parte del proyecto. Actualmente, contiene la función `main()`. Reemplace su contenido por lo siguiente:

```
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;  
}
```

La instrucción `import BasicPlane.Figures;` hace que todas las funciones y tipos exportados del `BasicPlane.Figures` módulo sean visibles para este archivo. Puede venir antes o después de cualquier `#include` directiva.

A continuación, la aplicación usa los tipos y funciones del módulo para generar como salida el área y el ancho del rectángulo definido:

Resultados

```
area: 50  
width: 10
```

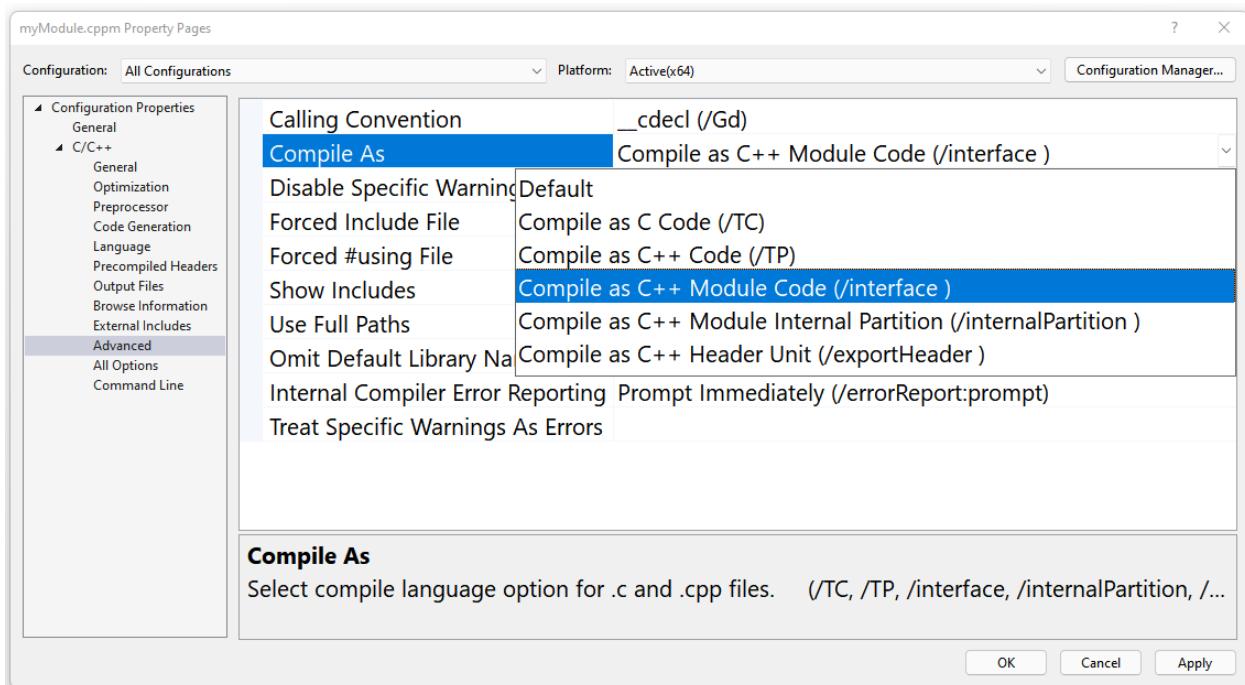
Anatomía de un módulo

Ahora, veamos con más detalle los distintos archivos de módulo.

Interfaz principal del módulo

Un módulo consta de uno o varios archivos. Uno de ellos define la interfaz que verán los importadores. Este archivo contiene la *interfaz principal del módulo*. Solo puede haber una interfaz principal del módulo por cada módulo. Como se indicó anteriormente, la unidad de interfaz del módulo exportado no especifica una partición de módulo.

Tiene la extensión `.ixx` de manera predeterminada. Sin embargo, puede tratar un archivo de código fuente con cualquier extensión como un archivo de interfaz de módulo. Para ello, establezca la propiedad **Compilar como** en la pestaña **Opciones avanzadas** de la página de propiedades del archivo de origen en **Compilar como módulo (/interfaz):**



El esquema básico de un archivo de definición de interfaz de módulo es el siguiente:

```
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.
```

Este archivo debe comenzar con `module;` para indicar el principio del fragmento global del módulo o con `export module [module-name];` para indicar el inicio del *ámbito del módulo*.

El módulo purview es donde las funciones, tipos, plantillas, etc., van a exponer desde el módulo.

También es donde puede exponer otros módulos o particiones de módulo mediante las palabras clave `export` `import`, como se muestra en el archivo `BasicPlane.Figures.ixx`.

El archivo de interfaz principal debe exportar todas las particiones de interfaz definidas para el módulo directa o indirectamente, o el programa tiene un formato incorrecto.

La partición privada del módulo es donde puede colocar los elementos que desea que solo sean visibles en este archivo.

Las unidades de interfaz de módulo usan como prefijo de la palabra clave `module` la palabra clave `export`.

Para obtener una visión más detallada de la sintaxis del módulo, consulte [Introducción a los módulos en C++](#).

Unidades de implementación de módulo

Las unidades de implementación de módulo pertenecen a un módulo con nombre. El módulo con nombre al que pertenecen se indica mediante la `module [module-name]` instrucción del archivo . Las unidades de implementación de módulo proporcionan detalles de implementación que, por motivos de higiene del código u otros motivos, no desea colocar en la interfaz principal del módulo ni en un archivo de partición de módulo.

Las unidades de implementación de módulos son útiles para dividir un módulo grande en partes más pequeñas, lo que puede dar lugar a tiempos de compilación más rápidos. Esta técnica se trata brevemente en la sección [Procedimientos recomendados para módulos](#).

Los archivos de unidad de implementación de módulo tienen la extensión `.cpp`. El esquema básico de un archivo de unidad de implementación de módulo es el siguiente:

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
```

Archivos de partición de módulo

Las particiones de módulo proporcionan una manera de crear componentes de un módulo en diferentes partes o *particiones*. Las particiones de módulo están pensadas para su importación solo en archivos que forman parte del módulo con nombre. No se pueden importar fuera del módulo con nombre.

Una partición tiene un archivo de interfaz y cero o más archivos de implementación. Una partición de módulo comparte la propiedad de todas las declaraciones del módulo completo.

Todos los nombres exportados por los archivos de interfaz de partición son importados y reexportados (`export import`) por el archivo de la interfaz principal. El nombre de una partición debe comenzar con el nombre del módulo, seguido de dos puntos y, a continuación, el nombre de la partición.

El esquema básico de un archivo de interfaz de partición tiene este aspecto:

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.

...

Procedimientos recomendados del módulo

Un módulo y el código que lo importa se deben compilar con las mismas opciones del compilador.

Nomenclatura de los módulos

- Puede usar puntos (".") en los nombres de módulo, pero no tienen ningún significado especial para el compilador. Úselos para transmitir el significado a los usuarios del módulo. Por ejemplo, comience con el espacio de nombres de nivel superior del proyecto o la biblioteca. Finalice con un nombre que describa la funcionalidad del módulo. `BasicPlane.Figures` está pensado para transmitir una API para planos geométricos, y específicamente figuras que se pueden representar en un plano.
- El nombre del archivo que contiene la interfaz principal del módulo suele ser el nombre del módulo. Por ejemplo, dado el nombre de módulo `BasicPlane.Figures`, el nombre del archivo que contiene la interfaz principal se llamaría `BasicPlane.Figures.ixx`.
- El nombre de un archivo de partición de módulo suele ser `<primary-module-name>-<module-partition-name>`, donde el nombre del módulo va seguido de un guión ("-") y, a continuación, el nombre de la partición. Por ejemplo: `BasicPlane.Figures-Rectangle.ixx`

Si va a compilar desde la línea de comandos y usa esta convención de nomenclatura para las particiones de módulo, no tendrá que agregar explícitamente `/reference` para cada archivo de partición de módulo. El compilador los buscará automáticamente en función del nombre del módulo. El nombre del archivo de partición compilado (que termina con una `.ifc` extensión) se genera a partir del nombre del módulo. Considere el nombre `BasicPlane.Figures:Rectangle` del módulo: el compilador prevé que el archivo de partición compilado correspondiente para `Rectangle` se denomina `BasicPlane.Figures-Rectangle.ifc`. El compilador usa este esquema de nomenclatura para facilitar el uso de particiones de módulo mediante la búsqueda automática de los archivos de unidad de interfaz para las particiones.

Puede asignarles un nombre mediante su propia convención. Sin embargo, deberá especificar los argumentos `/reference` correspondientes al compilador de la línea de comandos.

Módulos de factor

Use archivos de implementación de módulo y particiones para factorizar el módulo para facilitar el mantenimiento del código y tener tiempos de compilación potencialmente más rápidos.

Por ejemplo, mover la implementación de un módulo fuera del archivo de definición de interfaz de módulo y a un archivo de implementación de módulo significa que los cambios en la implementación no harán necesariamente que todos los archivos que importan el módulo se vuelvan a compilar (a menos que tenga implementaciones `inline`).

Las particiones de módulo facilitan la factorización lógica de un módulo grande. Se pueden usar para mejorar el tiempo de compilación para que los cambios realizados en una parte de la implementación no provoquen que se vuelvan a compilar todos los archivos del módulo.

Resumen

En este tutorial, se han presentado los conceptos básicos de los módulos de C++20. Ha creado una interfaz principal del módulo, ha definido una partición de módulo y ha creado un archivo de implementación de módulo.

Consulte también

[Información general de los módulos en C++](#)

[Palabras clave module, import y export](#)

[Un recorrido por los módulos de C++ en Visual Studio ↗](#)

[Módulos de C++20 en la práctica y el futuro de las herramientas en torno a los módulos de C++ ↗](#)

[Traslado de un proyecto a módulos con nombre de C++ ↗](#)

[Tutorial: Compilación e importación de unidades de encabezado en Microsoft Visual C++ ↗](#)

Plantillas (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 7 minutos

Las plantillas son la base para la programación genérica en C++. Como lenguaje fuertemente tipado, C++ requiere que todas las variables tengan un tipo específico, ya sea declarado explícitamente por el programador o deducido por el compilador. Sin embargo, muchas estructuras de datos y algoritmos tienen el mismo aspecto independientemente del tipo en el que operan. Las plantillas permiten definir las operaciones de una clase o función y permiten al usuario especificar en qué tipos concretos deben funcionar esas operaciones.

Definición y uso de plantillas

Una plantilla es una construcción que genera un tipo o función normal en tiempo de compilación en función de los argumentos que proporciona el usuario para los parámetros de la plantilla. Por ejemplo, se puede definir una plantilla de función como esta:

```
C++  
  
template <typename T>  
T minimum(const T& lhs, const T& rhs)  
{  
    return lhs < rhs ? lhs : rhs;  
}
```

En el código anterior se describe una plantilla para una función genérica con un único parámetro de tipo T , cuyo valor devuelto y parámetros de llamada (`lhs` y `rhs`) son todos de este tipo. Se puede asignar un nombre a un parámetro de tipo como se desee, pero por convención se usan letras mayúsculas únicas. T es un parámetro de plantilla; la palabra clave `typename` dice que este parámetro es un marcador de posición para un tipo. Cuando se llama a la función, el compilador reemplazará cada instancia de `T` por el argumento de tipo concreto especificado por el usuario o deducido por el compilador. El proceso en el que el compilador genera una clase o función a partir de una plantilla se conoce como *creación de una instancia de plantilla*; `minimum<int>` es una creación de instancias de la plantilla `minimum<T>`.

En otro lugar, un usuario puede declarar una instancia de la plantilla especializada para `int`. Suponiéndose que `get_a()` y `get_b()` son funciones que devuelven un valor `int`:

```
C++
```

```
int a = get_a();
int b = get_b();
int i = minimum<int>(a, b);
```

Sin embargo, dado que se trata de una plantilla de función y el compilador puede deducir el tipo `T` de los argumentos `a` y `b`, se puede llamar igual que una función normal:

C++

```
int i = minimum(a, b);
```

Cuando el compilador encuentra esa última instrucción, genera una nueva función en la que cada aparición de `T` en la plantilla se reemplaza por `int`:

C++

```
int minimum(const int& lhs, const int& rhs)
{
    return lhs < rhs ? lhs : rhs;
}
```

Las reglas de cómo el compilador realiza la deducción de tipos en las plantillas de función se basan en las reglas de las funciones normales. Para obtener más información, consulte [Resolución de sobrecarga de las llamadas de la plantilla de función](#).

Parámetros de tipo

En la plantilla `minimum` anterior, tenga en cuenta que el parámetro de tipo `T` no está calificado de ninguna manera hasta que se usa en los parámetros de llamada de función, donde se agregan los calificadores de referencia y `const`.

No hay ningún límite práctico para el número de parámetros de tipo genérico. Separe los distintos parámetros con comas:

C++

```
template <typename T, typename U, typename V> class Foo{};
```

La palabra clave `class` es equivalente a `typename` en este contexto. Puede expresar el ejemplo anterior como:

C++

```
template <class T, class U, class V> class Foo{};
```

Puede usar el operador de puntos suspensivos (...) para definir una plantilla que toma un número arbitrario de cero o más parámetros de tipo:

C++

```
template<typename... Arguments> class vtclass;

vtclass<> vtinstance1;
vtclass<int> vtinstance2;
vtclass<float, bool> vtinstance3;
```

Cualquier tipo integrado o definido por el usuario se puede usar como un argumento de tipo. Por ejemplo, se puede usar `std::vector` en la Biblioteca Estándar para almacenar las variables de tipo `int`, `double`, `std::string`, `MyClass`, `const MyClass*`, `MyClass&`, etc. La restricción principal al usar plantillas es que un argumento de tipo debe admitir las operaciones que se aplican a los parámetros de tipo. Por ejemplo, si llamamos a `minimum` usando `MyClass` como en este ejemplo:

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
}
```

Se generará un error del compilador porque `MyClass` no proporciona una sobrecarga para el operador `<`.

No hay ningún requisito inherente de que los argumentos de tipo para cualquier plantilla determinada pertenezcan a la misma jerarquía de objetos, aunque se puede definir una plantilla que aplique dicha restricción. Se pueden combinar técnicas orientadas a objetos con plantillas; por ejemplo, se puede almacenar un objeto `Derived*` en un vector`<Base*>`. Tenga en cuenta que los argumentos deben ser punteros

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>());
```

Los requisitos básicos que `std::vector` y otros contenedores de la biblioteca estándar imponen a los elementos de `T`, es que estos `T` se puedan copiar y construir.

Parámetros que no son de tipo

A diferencia de los tipos genéricos de otros lenguajes, como C# y Java, las plantillas de C++ admiten *parámetros que no son de tipo* que también son denominados parámetros de valor. Por ejemplo, se puede proporcionar un valor entero constante para especificar la longitud de una matriz, como en este ejemplo similar a la clase `std::array` de la Biblioteca Estándar:

C++

```
template<typename T, size_t L>
class MyArray
{
    T arr[L];
public:
    MyArray() { ... }
};
```

Tenga en cuenta la sintaxis de la declaración de la plantilla. El valor `size_t` se pasa como argumento de la plantilla en tiempo de compilación y debe ser `const` o una expresión `constexpr`. Se usa de la siguiente manera:

C++

```
MyArray<MyClass*, 10> arr;
```

Otros tipos de valores, incluidos punteros y referencias, se pueden pasar como parámetros que no son de tipo. Por ejemplo, se puede pasar un puntero a una función o un objeto de función para personalizar alguna operación dentro del código de plantilla.

Deducción de tipos para los parámetros de plantilla que no son de tipo

En Visual Studio 2017 y versiones posteriores, además del modo `/std:c++17` o posterior, el compilador deduce el tipo de un argumento de la plantilla que no es de tipo declarado con `auto`:

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
```

Plantillas como parámetros de plantilla

Una plantilla puede ser un parámetro de plantilla. En este ejemplo, `MyClass2` tiene dos parámetros de plantilla: un parámetro `typename T` y un parámetro de plantilla `Arr`:

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
};
```

Dado que el propio parámetro `Arr` no tiene cuerpo, sus nombres de parámetro no son necesarios. De hecho, es un error hacer referencia a los nombres de parámetro de clase o el `typename` de `Arr` desde el cuerpo de `MyClass2`. Por este motivo, se pueden omitir los nombres de parámetros de tipo `Arr`, como se muestra en este ejemplo:

C++

```
template<typename T, template<typename, int> class Arr>
class MyClass2
{
    T t; //OK
    Arr<T, 10> a;
};
```

Argumentos de plantilla predeterminados

Las plantillas de clase y función pueden tener argumentos predeterminados. Cuando una plantilla tiene un argumento predeterminado, se puede dejar sin especificar al usarlo. Por ejemplo, la plantilla std::vector tiene un argumento predeterminado para el asignador:

C++

```
template <class T, class Allocator = allocator<T>> class vector;
```

En la mayoría de los casos, la clase std::allocator predeterminada es aceptable, por lo que se usa un vector similar al siguiente:

C++

```
vector<int> myInts;
```

Pero si es necesario, se puede especificar un asignador personalizado de la siguiente manera:

C++

```
vector<int>, MyAllocator> ints;
```

Para varios argumentos de plantilla, todos los argumentos después del primer argumento predeterminado deben tener argumentos predeterminados.

Cuando se usa una plantilla cuyos parámetros están todos predeterminados, se usan corchetes angulares vacíos:

C++

```
template<typename A = int, typename B = double>
class Bar
{
    //...
};

...
int main()
{
    Bar<> bar; // use all default type arguments
}
```

Especialización de plantilla

En algunos casos, no es posible ni deseable que una plantilla defina exactamente el mismo código para cualquier tipo. Por ejemplo, es posible que se desee definir una ruta de acceso de código que se va a ejecutar solo si el argumento de tipo es un puntero o un std::wstring, o un tipo derivado de una clase base determinada. En tales casos, se puede definir una *especialización* de la plantilla para ese tipo determinado. Cuando un usuario crea una instancia de la plantilla con ese tipo, el compilador usa la especialización para generar la clase y para todos los demás tipos, el compilador elige la plantilla más general. Las especializaciones en las que todos los parámetros están especializados son *especializaciones completas*. Si solo algunos de los parámetros están especializados, se denomina *especialización parcial*.

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
```

Una plantilla puede tener cualquier número de especializaciones siempre que cada parámetro de tipo especializado sea único. Solo las plantillas de clase pueden estar parcialmente especializadas. Todas las especializaciones completas y parciales de una plantilla deben declararse en el mismo espacio de nombres que la plantilla original.

Para obtener más información, consulte [Especificaciones de plantilla](#).

typename

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

En las definiciones de plantilla, `typename` proporciona una sugerencia al compilador de que un identificador desconocido es un tipo. En las listas de parámetros de plantilla, se usa para especificar un parámetro de tipo.

Sintaxis

```
typename identifier ;
```

Comentarios

La `typename` palabra clave debe usarse si un nombre de una definición de plantilla es un nombre completo que depende de un argumento de plantilla; es opcional si el nombre completo no depende. Para obtener más información, consulte [Plantillas y resolución de nombres](#).

`typename` se puede usar en cualquier tipo y en cualquier lugar de una declaración o definición de plantilla. No se permite en la lista de clases base, a menos que sea un argumento de plantilla para una clase base de plantilla.

C++

```
template <class T>
class C1 : typename T::InnerType // Error - typename not allowed.
{};
template <class T>
class C2 : A<typename T::InnerType> // typename OK.
{};


```

La palabra clave `typename` también se puede usar en lugar de `class` en las listas de parámetros de plantilla. Por ejemplo, las instrucciones siguientes son semánticamente equivalentes:

C++

```
template<class T1, class T2>...
template<typename T1, typename T2>...
```

Ejemplo

C++

```
// typename.cpp
template<class T> class X
{
    typename T::Y m_y; // treat Y as a type
};

int main()
{}
```

Consulte también

[Templates \(Plantillas \[C++\]\)](#)

[Palabras clave](#)

Plantillas de clase

Artículo • 03/03/2023 • Tiempo de lectura: 7 minutos

En este artículo se describen las reglas específicas de las plantillas de clase de C++.

Funciones miembro de plantillas de clase

Las funciones miembro se pueden definir dentro o fuera de una plantilla de clase. Se definen como plantillas de función si se definen fuera de la plantilla de clase.

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()
{}
```

Al igual que con cualquier función miembro de clase de plantilla, la definición de la función miembro del constructor de la clase incluye la lista de argumentos de plantilla dos veces.

Las funciones miembro pueden ser plantillas de función y especificar parámetros adicionales, como en el ejemplo siguiente.

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()
{
}
```

Plantillas de clase anidadas

Las plantillas se pueden definir dentro de clases o plantillas de clase, en cuyo caso se conocen como plantillas de miembro. Las plantillas de miembro que son clases se conocen como plantillas de clase anidadas. Las plantillas de miembro que son funciones se tratan en [Plantillas de función miembro](#).

Las plantillas de clase anidada se declaran como plantillas de clase dentro del ámbito de la clase externa. Pueden definirse dentro o fuera de la clase envolvente.

En el código siguiente se muestra una plantilla de clase anidada dentro de una clase ordinaria.

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();
}

```

El código siguiente usa parámetros de tipo de plantilla anidada para crear plantillas de clase anidadas:

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()
*/

```

No se permite que las clases locales tengan plantillas de miembro.

Friends de plantilla

Las plantillas de clase pueden tener elementos **friend**. Una clase o plantilla de clase y una función o plantilla de función pueden ser elementos friend de una clase de plantilla. Los elementos friend también pueden ser especializaciones de una plantilla de clase o de función, pero no especializaciones parciales.

En el ejemplo siguiente, una función friend se define como una plantilla de función dentro de la plantilla de clase. Este código genera una versión de la función friend para cada instancia de la plantilla. Esta construcción es útil si la función friend depende de los mismos parámetros de plantilla que la clase.

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
*/

```

En el ejemplo siguiente se incluye una función friend que tiene una especialización de plantilla. Una especialización de plantilla de función es automáticamente friend si la plantilla de función original es friend.

También se puede declarar solo la versión especializada de la plantilla como friend, como indica el comentario que precede a la declaración friend del código siguiente. Si declara una especialización como un friend, debe colocar la definición de especialización de plantilla friend fuera de la clase de plantilla.

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
*/

```

En el ejemplo siguiente se muestra una plantilla de clase friend declarada dentro de una plantilla de clase. La plantilla de clase se usa como argumento de plantilla para la clase friend. Las plantillas de clase friend se deben definir fuera de la plantilla de clase en la que se declaran. Las especializaciones o especializaciones parciales de las plantillas friend son también elementos friend de la plantilla de clase original.

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
*/

```

Reutilizar parámetros de plantilla

Los parámetros de plantilla se pueden reutilizar en la lista de parámetros de plantilla. Por ejemplo, se permite el código siguiente:

```

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()
{
}

```

Consulte también

Templates (Plantillas [C++])

Plantillas de función

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Las plantillas de clase definen una familia de clases relacionadas que se basan en los argumentos de tipo pasados a la clase al crear instancias. Las plantillas de función son similares a las plantillas de clase, pero definen una familia de funciones. Con las plantillas de función, puede especificar un conjunto de funciones que se basen en el mismo código pero que actúen en diferentes tipos o clases. La siguiente plantilla de función intercambia dos elementos:

C++

```
// function_templates1.cpp
template< class T > void MySwap( T& a, T& b ) {
    T c(a);
    a = b;
    b = c;
}
int main() {
```

Este código define una familia de funciones que intercambian los valores de los argumentos. En esta plantilla, puede generar funciones que intercambien tipos `int` y `long`, así como tipos definidos por el usuario. `MySwap` intercambiaria incluso las clases si el constructor de copias y el operador de asignación de la clase se definen correctamente.

Además, la plantilla de función evitará que se intercambien objetos de diferentes tipos, ya que el compilador conoce los tipos de los parámetros *a* y *b* en tiempo de compilación.

Aunque esta función se puede ejecutar mediante una función sin plantilla, usando punteros void, la versión de plantilla proporciona seguridad de tipos. Observe las siguientes llamadas:

C++

```
int j = 10;
int k = 18;
CString Hello = "Hello, Windows!";
MySwap( j, k );           //OK
MySwap( j, Hello );      //error
```

La segunda llamada de `MySwap` origina un error en tiempo de compilación, porque el compilador no puede generar una función `MySwap` con parámetros de tipos diferentes. Si se usaran punteros void, ambas llamadas de función se compilarían correctamente, pero la función no funcionaría adecuadamente en tiempo de ejecución.

Se permite la especificación explícita de los argumentos de plantilla para una plantilla de función. Por ejemplo:

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.
}
```

Cuando el argumento de plantilla se especifica explícitamente, se realizan las conversiones implícitas normales para convertir el argumento de la función al tipo de los parámetros de la plantilla de función correspondientes. En el ejemplo anterior, el compilador convertirá `j` al tipo `char`.

Consulte también

[Templates \(Plantillas \[C++\]\)](#)

[Crear instancias de plantillas de función](#)

[Creación de instancias explícita](#)

[Especialización explícita de las plantillas de función](#)

Crear instancias de plantillas de función

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Cuando se llama por primera vez a una plantilla de función para cada tipo, el compilador genera una creación de instancias. Cada creación de instancias es una versión de la función con plantilla especializada para el tipo concreto. Se llamará a esta creación de instancias cada vez que se use la función para el tipo. Si tiene varias creaciones de instancias idénticas, incluso en módulos diferentes, solo una copia de la creación de instancias terminará agregándose al archivo ejecutable.

La conversión de argumentos de función se permite en las plantillas de función de cualquier par de argumento y parámetro en el que el parámetro no dependa de un argumento de plantilla.

Se pueden crear explícitamente instancias de las plantillas de función si la plantilla se declara con un tipo concreto como argumento. Por ejemplo, se permite el código siguiente:

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()
{}
```

Consulte también

[Plantillas de función](#)

Creación de instancias explícita

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Puede utilizar la creación de instancias explícita para crear una instancia de una clase o una función con plantilla sin usarla realmente en el código. Dado que resulta útil al crear archivos de biblioteca (`.lib`) que usan plantillas para la distribución, las definiciones de plantilla no inicializadas no se colocan en archivos de objeto (`.obj`).

Ejemplos

Este código crea explícitamente instancias de `MyStack` para las variables `int` y seis elementos:

C++

```
template class MyStack<int, 6>;
```

Esta instrucción crea una instancia de `MyStack` sin reservar ningún almacenamiento para un objeto. Se genera el código para todos los miembros.

La línea siguiente crea explícitamente instancias solo de la función miembro de constructor:

C++

```
template MyStack<int, 6>::MyStack( void );
```

Puede crear instancias explícitas de plantillas de función mediante un argumento de tipo específico para volver a declararlas, como se muestra en el ejemplo de creación de [instancias de plantilla de función](#).

Puede usar la palabra clave `extern` para impedir la creación automática de instancias de miembros. Por ejemplo:

C++

```
extern template class MyStack<int, 6>;
```

Del mismo modo, puede marcar determinados miembros como externos y no crear instancias de ellos:

C++

```
extern template MyStack<int, 6>::MyStack( void );
```

Puede usar la palabra clave `extern` para impedir que el compilador genere el mismo código de creación de instancias en más de un módulo de objeto. Debe crear una instancia de la plantilla de función mediante los parámetros de plantilla explícitos especificados en al menos un módulo vinculado si se llama a la función. De lo contrario, obtendrá un error del vinculador cuando se compila el programa.

ⓘ Nota

La palabra clave `extern` en la especialización solo se aplica a las funciones de miembro definidas fuera del cuerpo de la clase. Las funciones definidas dentro de la declaración de clase se consideran funciones insertadas y siempre se crean instancias de ellas.

Consulte también

[Plantillas de función](#)

Especialización explícita de las plantillas de función

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Con una plantilla de función, puede definir un comportamiento especial para un tipo específico si proporciona una especialización explícita (reemplazo) de la plantilla de función de ese tipo. Por ejemplo:

C++

```
template<> void MySwap(double a, double b);
```

Esta declaración permite definir una función diferente para las variables de tipo `double`. Al igual que en las funciones que no son de plantilla, se aplican las conversiones de tipo estándar (por ejemplo, promover una variable de tipo `float` a tipo `double`).

Ejemplo

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()
{}
```

Consulte también

Plantillas de función

Ordenación parcial de plantillas de función (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Puede haber varias plantillas de función que coincidan con la lista de argumentos de una llamada de función. C++ define el orden parcial de las plantillas de función para especificar a qué función se debe llamar. El orden es parcial porque puede haber algunas plantillas que se consideren especializadas por igual.

El compilador elige la plantilla de función más especializada disponible en las posibles coincidencias. Por ejemplo, si una plantilla de función toma un tipo T y hay otra plantilla de función que toma T^* , se considera que la versión de T^* es más especializada. Se prefiere esta antes que la versión genérica T siempre que el argumento sea un tipo de puntero, aunque ambas coincidencias estarían permitidas.

Utilice el proceso siguiente para determinar si un candidato de plantilla de función es más especializado:

1. Considere dos plantillas de función y $T_1 T_2$.
2. Reemplace los parámetros de por T_1 un tipo x único hipotético .
3. Con la lista de parámetros en T_1 , consulte si T_2 es una plantilla válida para esa lista de parámetros. Omita cualquier conversión implícita.
4. Repita el mismo proceso con T_1 e T_2 invertido.
5. Si una plantilla es una lista de argumentos de plantilla válida para otra plantilla, pero no al revés, esa plantilla se considera menos especializada que la otra plantilla. Si, con el paso anterior, las dos plantillas forman argumentos válidos para ambas, se consideran especializadas por igual y, cuando se intente usarlas, se producirá una llamada ambigua.
6. Según estas reglas:
 - a. Una especialización de plantilla para un tipo concreto se considera más especializada que una que toma un argumento de tipo genérico.
 - b. Una plantilla que toma solo T^* es más especializada que una que toma solo T , porque un tipo x^* hipotético es un argumento válido para un T argumento de plantilla, pero x no es un argumento válido para un T^* argumento de plantilla.

- c. `const T` es más especializado que `T`, porque `const X` es un argumento válido para un `T` argumento de plantilla, pero `X` no es un argumento válido para un `const T` argumento de plantilla.
- d. `const T*` es más especializado que `T*`, porque `const X*` es un argumento válido para un `T*` argumento de plantilla, pero `X*` no es un argumento válido para un `const T*` argumento de plantilla.

Ejemplo

El ejemplo siguiente funciona según se especifica en el estándar:

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.
}
```

Salida

Output

Less specialized function called
More specialized function called
Even more specialized function for const T*

Consulte también

[Plantillas de función](#)

Plantillas de función miembro

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

El término plantilla de miembro se refiere tanto a las plantillas de función miembro como a las plantillas de clase anidada. Las plantillas de función miembro son plantillas de función que son miembros de una plantilla de clase o clase.

Las funciones miembro pueden ser plantillas de función en varios contextos. Todas las funciones de las plantillas de clase son genéricas, pero no se conocen como plantillas de miembro o plantillas de función miembro. Si estas funciones miembro toman sus propios argumentos de plantilla, se consideran plantillas de función miembro.

Ejemplo: Declaración de plantillas de función miembro

Las plantillas de función miembro de clases que no son de plantilla o plantillas de clase se declaran como plantillas de función con sus parámetros de plantilla.

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);
}
```

Ejemplo: Plantilla de función miembro de una plantilla de clase

En el ejemplo siguiente se muestra una plantilla de función miembro de una plantilla de clase.

C++

```
// member_function_templates2.cpp
template<typename T>
class X
{
public:
    template<typename U>
    void mf(const U &u)
    {
    }
};

int main()
{
}
```

Ejemplo: Definición de plantillas de miembro fuera de una clase

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()
{}
```

Ejemplo: Conversión con plantilla definida por el usuario

No se permite que las clases locales tengan plantillas de miembro.

Las plantillas de función miembro no pueden ser funciones virtuales. Además, no pueden invalidar las funciones virtuales de una clase base cuando se declaran con el

mismo nombre que una función virtual de clase base.

En el ejemplo siguiente se muestra una conversión definida por el usuario con plantilla:

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>.
}
```

Consulte también

[Plantillas de función](#)

Especialización de plantilla (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 5 minutos

Las plantillas de clase pueden especializarse parcialmente y la clase resultante es una plantilla. La especialización parcial permite que el código de plantilla se personalice parcialmente para tipos específicos en situaciones como las siguientes:

- Una plantilla tiene varios tipos y solo algunos de ellos deben estar especializados. El resultado es una plantilla con parámetros en los tipos restantes.
- Una plantilla solo tiene un tipo, pero una especialización es necesaria para el puntero, la referencia, el puntero a miembro o los tipos de puntero a función. La propia especialización sigue siendo una plantilla en el tipo al que señala o al que hace referencia.

Ejemplo: Especialización parcial de plantillas de clase

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

```

Ejemplo: Especialización parcial para tipos de puntero

Si tiene una clase de colección de plantillas que toma cualquier tipo `T`, puede crear una especialización parcial que toma cualquier tipo de puntero `T*`. En el código siguiente se muestra una plantilla de clase de colección `Bag` y una especialización parcial para los tipos de puntero en los que la colección desreferencia los tipos de puntero antes de copiarlos en la matriz. A continuación, la colección almacena los valores a los que se señala. Con la plantilla original, solo los propios punteros se hubieran almacenado en la colección y los datos serían vulnerables a la eliminación o la modificación. En esta versión de puntero especial de la colección, se agrega código para comprobar si hay un puntero `NULL` en el método `add`.

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

```

Ejemplo: Definir una especialización parcial para que un tipo sea `int`

En el ejemplo siguiente se define una clase de plantilla que toma pares de dos tipos cualquiera y después define una especialización parcial de esa clase de plantilla especializada de modo que uno de los tipos sea `int`. La especialización define un

método de ordenación adicional que implementa una ordenación de burbuja simple basada en el entero.

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}
```

Plantillas y resolución de nombres

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

En las definiciones de plantilla, hay tres tipos de nombres.

- Nombres declarados localmente, como el nombre de la propia plantilla y cualquier nombre declarado dentro de la definición de plantilla.
- Nombres del ámbito de inclusión fuera de la definición de plantilla.
- Nombres que dependen de alguna manera de los argumentos de plantilla, denominados nombres dependientes.

Mientras que los dos primeros nombres pertenecen también a ámbitos de clase y función, en las definiciones de plantillas se requieren reglas especiales de resolución de nombres para tratar la complejidad agregada de los nombres dependientes. Esto se debe a que el compilador apenas tiene conocimiento de estos nombres hasta que se crea una instancia de la plantilla, ya que pueden ser tipos totalmente diferentes en función de los argumentos de plantilla que se usen. Los nombres no dependientes se buscan de acuerdo con las reglas habituales y en el punto de definición de la plantilla. Estos nombres, que son independientes de los argumentos de plantilla, se buscan una sola vez para todas las especializaciones de plantilla. Los nombres dependientes no se buscan hasta que se crea una instancia de la plantilla y se buscan por separado para cada especialización.

Un tipo es dependiente si depende de los argumentos de plantilla. En concreto, un tipo es dependiente si es:

- El propio argumento de plantilla:

C++

T

- Un nombre completo con una clasificación que incluye un tipo dependiente:

C++

T::myType

- Un nombre completo si la parte incompleta identifica un tipo dependiente:

C++

N::T

- Un tipo const o volatile para el que el tipo base es un tipo dependiente:

C++

```
const T
```

- Un tipo de puntero, referencia, matriz o puntero de función basado en un tipo dependiente:

C++

```
T *, T &, T [10], T (*)()
```

- Una matriz cuyo tamaño se basa en un parámetro de plantilla:

C++

```
template <int arg> class X {  
    int x[arg]; // dependent type  
}
```

- Un tipo de plantilla creado a partir de un parámetro de plantilla:

C++

```
T<int>, MyTemplate<T>
```

Dependencia de tipos y dependencia de valores

Los nombres y las expresiones dependientes de un parámetro de plantilla se clasifican como dependientes del tipo o dependientes del valor, en función de si el parámetro de plantilla es un parámetro de tipo o un parámetro de valor. Además, cualquier identificador declarado en una plantilla con un tipo dependiente del argumento de plantilla se considera dependiente del valor, como en el caso de un tipo de entero o de enumeración inicializado con una expresión dependiente del valor.

Las expresiones dependientes del tipo y dependientes del valor son expresiones que implican variables dependientes del tipo o dependientes del valor. Estas expresiones

pueden tener una semántica diferente en función de los parámetros utilizados para la plantilla.

Consulte también

[Templates \(Plantillas \[C++\]\)](#)

Resolución de nombres para los tipos dependientes

Artículo • 26/09/2022 • Tiempo de lectura: 2 minutos

Utilice `typename` para los nombres completos en definiciones de plantilla, para indicar al compilador que el nombre completo dado identifica un tipo. Para obtener más información, vea [typename](#).

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.
```

La búsqueda de nombres para nombres dependientes examina los nombres tanto del contexto de la definición de la plantilla (en el siguiente ejemplo, este contexto buscaría `MyNamespace::myFunction(char)`) como del contexto de la creación de una instancia de la plantilla. En el siguiente ejemplo, la plantilla se instancia en `main`; por lo tanto, `MyNamespace::myFunction` es visible desde el punto de creación de instancias y se elige como la mejor coincidencia. Si `MyNamespace::myFunction` cambiase de nombre, se llamaría a `myFunction(char)` en su lugar.

Todos los nombres se resuelven como si fueran nombres dependientes. Sin embargo, recomendamos utilizar nombres completos si hay alguna posibilidad de conflicto.

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);
}
```

Output

Output

```
Int MyNamespace::myFunction
```

Desambiguación de plantilla

Visual Studio 2012 aplica las reglas estándar de C++98/03/11 para la eliminación de ambigüedades con la palabra clave "plantilla". En el ejemplo siguiente, Visual Studio 2010 aceptaría las líneas no conformes y las líneas conformes. Visual Studio 2012 acepta solo las líneas conformes.

```
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;  
}
```

Se requiere la conformidad con las reglas de desambiguación, porque, de forma predeterminada, C++ supone que `AY::Rebind` no es una plantilla, por lo que el compilador interpreta el siguiente "`<`" como "menos que". Debe saber que `Rebind` es una plantilla para que pueda analizar correctamente "`<`" como un corchete angular.

Consulte también

[Resolución de nombres](#)

Resolución de nombres declarados localmente

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

Se puede hacer referencia al propio nombre de la pantalla con o sin argumentos de plantilla. En el ámbito de una plantilla de clase, el nombre en sí hace referencia a la plantilla. En el ámbito de una especialización de plantilla o especialización parcial, el nombre hace referencia a la especialización o especialización parcial. También se puede hacer referencia a otras especializaciones o especializaciones parciales de la plantilla con los argumentos de plantilla adecuados.

Ejemplo: especialización frente a la especialización parcial

El código siguiente muestra que el nombre `A` de la plantilla de clase se interpreta de forma diferente en el ámbito de una especialización o especialización parcial.

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>.
};
```

Ejemplo: conflicto de nombres entre el parámetro de plantilla y el objeto

Cuando hay un conflicto de nombres entre un parámetro de plantilla y otro objeto, el parámetro de plantilla puede o no se puede ocultar. Las reglas siguientes ayudarán a determinar la prioridad.

El parámetro de plantilla está dentro del ámbito desde el punto donde aparece por primera vez hasta el final de la plantilla de clase o función. Si el nombre aparece de nuevo en la lista de argumentos de plantilla o en la lista de clases base, hace referencia al mismo tipo. En C++ estándar, no se puede declarar ningún otro nombre que sea idéntico al parámetro de plantilla en el mismo ámbito. Una extensión de Microsoft permite que el parámetro de plantilla se vuelva a definir en el ámbito de la plantilla. En el ejemplo siguiente se muestra cómo usar el parámetro de plantilla en la especificación base de una plantilla de clase.

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;
}
```

Ejemplo: definición de una función miembro fuera de la plantilla de clase

Cuando las funciones miembro se definen fuera de la plantilla de clase, se puede usar un nombre de parámetro de plantilla diferente. Si la definición de la función miembro de plantilla de clase usa un nombre diferente para el parámetro de plantilla que la declaración, y el nombre usado en la definición entra en conflicto con otro miembro de la declaración, el miembro de la declaración de plantilla tiene prioridad.

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()
```

Ejemplo: definición de una plantilla o una función miembro fuera del espacio de nombres

Al definir una plantilla de función o una función miembro fuera del espacio de nombres en el que se declaró la plantilla, el argumento template tiene prioridad sobre los nombres de otros miembros del espacio de nombres.

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
```

Ejemplo: clase base o nombre de miembro oculta el argumento de plantilla

En las definiciones que están fuera de la declaración de clase de plantilla, si una clase de plantilla tiene una clase base que no depende de un argumento de plantilla y si la clase base o uno de sus miembros tienen el mismo nombre que un argumento de plantilla, la clase base o el nombre de miembro oculta el argumento de plantilla.

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

Consulte también

[Resolución de nombres](#)

Resolución de sobrecarga de llamadas de plantilla de función

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Una plantilla de función puede sobrecargar funciones que no son de plantilla con el mismo nombre. En este escenario, el compilador primero intenta resolver una llamada de función mediante la deducción de argumentos de plantilla para crear una instancia de la plantilla de función con una especialización única. Si se produce un error en la deducción de argumentos de plantilla, el compilador considera las sobrecargas de la plantilla de función con instancias y las sobrecargas de función que no son de plantilla para resolver la llamada. Estas otras sobrecargas se conocen como el *conjunto de candidatos*. Si la deducción del argumento de plantilla se realiza correctamente, la función generada se compara con las demás funciones del conjunto candidato para determinar la mejor coincidencia, siguiendo las reglas para la resolución de sobrecargas. Para obtener más información, consulte [Sobrecarga de funciones](#).

Ejemplo: Elección de una función que no es de plantilla

Si una función que no es de plantilla es una coincidencia igualmente buena con una plantilla de función, se elige la función que no es de plantilla (a menos que se especifiquen explícitamente los argumentos de plantilla), como en la llamada `f(1, 1)` en el ejemplo siguiente.

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)
```

Ejemplo: Plantilla de función de coincidencia exacta preferida

En el ejemplo siguiente se muestra que se prefiere la plantilla de función que coincide exactamente si la función que no es de plantilla requiere una conversión.

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)
```

Consulte también

Resolución de nombres

typename

Organización de código fuente (plantillas de C++)

Artículo • 03/03/2023 • Tiempo de lectura: 4 minutos

Al definir una plantilla de clase, debe organizar el código fuente de tal manera que las definiciones de miembro sean visibles para el compilador cuando las necesite. Tiene la opción de utilizar el *modelo de inclusión* o *creación de instancias explícita*. En el modelo de inclusión, se incluyen las definiciones de miembro en todos los archivos que usa una plantilla. Este enfoque es más sencillo y proporciona la máxima flexibilidad en cuanto a qué tipos concretos se pueden usar con la plantilla. La desventaja es que pueden aumentar los tiempos de compilación. El tiempo puede ser significativo si un proyecto o los archivos incluidos tienen un gran tamaño. Con el enfoque de creación de instancias explícita, la propia plantilla crea instancias de clases concretas o miembros de clase para tipos específicos. Este enfoque puede acelerar los tiempos de compilación, pero limita el uso a solo las clases que ha habilitado el implementador de plantilla antes de tiempo. En general, se recomienda usar el modelo de inclusión, a menos que los tiempos de compilación se conviertan en un problema.

Fondo

Las plantillas no son similares a las clases normales en el sentido de que el compilador no genera código de objeto para una plantilla o cualquiera de sus miembros. No se genera nada hasta que se crea una instancia de la plantilla con tipos concretos. Cuando el compilador encuentra una instancia de plantilla como `MyClass<int> mc;` y aún no existe ninguna clase con esa firma, genera una nueva clase. También intenta generar código para las funciones miembro que se usan. Si esas definiciones se encuentran en un archivo que no está incluido, directa o indirectamente, en el archivo .cpp que se está compilando, el compilador no puede verlas. Desde el punto de vista del compilador, no es necesariamente un error. Las funciones se pueden definir en otra unidad de traducción donde el enlazador las encontrará. Si el enlazador no encuentra ese código, genera un error *externo sin resolver*.

El modelo de inclusión

La manera más sencilla y habitual para hacer visible las definiciones de plantilla en una unidad de traducción es poner las definiciones en el propio archivo de encabezado. Cualquier archivo `.cpp` que usa la plantilla solo tiene que incluir el encabezado `#include`. Este es el enfoque usado en la biblioteca estándar.

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
```

Con este enfoque, el compilador tiene acceso a la definición de plantilla completa y puede crear instancias de plantillas y a petición para cualquier tipo. Es sencillo y relativamente fácil de mantener. Sin embargo, el modelo de inclusión tiene un costo en términos de tiempos de compilación. Este costo puede ser considerable en programas de gran tamaño, especialmente si el encabezado de plantilla propio incluye otros encabezados. Cada archivo `.cpp` que incluye el encabezado obtendrá su propia copia de las plantillas de función y todas las definiciones. El enlazador suele ser capaz de solucionar los problemas para que no se generen varias definiciones para una función, pero se necesita tiempo para hacer este trabajo. En los programas más pequeños, ese tiempo de compilación adicional probablemente no es significativo.

El modelo de creación de instancias explícita

Si el modelo de inclusión no es viable para el proyecto y sabe con certeza el conjunto de tipos que se usará para crear una instancia de una plantilla, puede separar el código de plantilla en un archivo `.h` y `.cpp`, y en el archivo `.cpp` crear explícitamente instancias de las plantillas. Este enfoque genera el código objeto que el compilador verá cuando encuentra las creaciones de instancias de usuario.

Cree una instancia explícita mediante el uso de la palabra clave `template` seguida por la firma de la entidad de la que desea crear una instancia. Esta entidad puede ser un tipo o un miembro. Si crea una instancia de un tipo explícitamente, se crean instancias de todos los miembros.

El archivo de encabezado `MyArray.h` declara la clase de plantilla `MyArray`:

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
```

El archivo de origen `MyArray.cpp` crea instancias explícitamente de `template MyArray<double, 5>` y `template MyArray<string, 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>;
```

En el ejemplo anterior, la creación de instancias explícita está en la parte inferior del archivo `.cpp`. `MyArray` puede utilizarse solo para tipos `double` o `String`.

ⓘ Nota

En C ++11, la palabra clave `export` ha quedado en desuso en el contexto de las definiciones de plantilla. En términos prácticos, esto tiene poco impacto porque la mayoría de los compiladores nunca son compatibles.

Control de eventos

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

El control de eventos se admite principalmente para las clases COM (las clases de C++ que implementan objetos COM, normalmente mediante clases ATL o el atributo [coclass](#)). Para obtener más información, vea [Control de eventos en COM](#).

El control de eventos también se admite para las clases C++ nativas (clases C++ que no implementan objetos COM). El soporte de control de eventos en C++ nativo está en desuso y se quitará en una versión futura. Para obtener más información, vea [Control de eventos en C++ nativo](#).

ⓘ Nota

Los atributos de eventos en C++ nativo no son compatibles con C++ estándar. No se compilan al especificar el modo de conformidad `/permissive-`.

El control de eventos admite el uso de uno y varios multiprocesos. Protege los datos del acceso simultáneo de multiprocesos. Se pueden衍生 subclases de las clases de origen o del receptor de eventos. Estas subclases admiten la obtención y recepción de eventos extendidos.

El compilador Microsoft C++ incluye atributos y palabras clave para declarar eventos y controladores de eventos. Los atributos y las palabras clave de eventos se pueden utilizar en programas de CLR y en programas de C++ nativo.

| Artículo | Descripción |
|--------------------------------|--|
| event_source | Crea un origen de eventos. |
| event_receiver | Crea un receptor de eventos (receptor). |
| _event | Declara un evento. |
| _raise | Resalta el sitio de llamada de un evento. |
| _hook | Asocia un método de control a un evento. |
| _unhook | Desvincula un método controlador de un evento. |

Consulte también

Referencia del lenguaje C++

Palabras clave

palabra clave `_event`

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

Declara un evento.

ⓘ Nota

Los atributos de eventos en C++ nativo no son compatibles con C++ estándar. No se compilan al especificar el modo de conformidad `/permissive-`.

Sintaxis

```
_event member-function-declarator ;
_event __interface interface-specifier ;
_event data-member-declarator ;
```

Comentarios

La palabra clave específica de Microsoft `_event` se puede aplicar a una declaración de función miembro, una declaración de interfaz o una declaración de miembro de datos. Sin embargo, no se puede usar la palabra clave `_event` para calificar a un miembro de una clase anidada.

Dependiendo de si el origen y el receptor del evento son C++ nativo, COM o administrados (.NET Framework), puede usar las siguientes construcciones como eventos:

| C++ nativo | COM | Administrada (.NET Framework) |
|-----------------|----------|-------------------------------|
| función miembro | - | method |
| - | interfaz | - |
| - | - | miembro de datos |

Use `_hook` en un receptor de eventos para asociar una función miembro de controlador a una función miembro de evento. Después de crear un evento con la palabra clave `_event`, se llama a todos los controladores de eventos asociados a ese evento después cuando se llama al evento.

Una función miembro `_event` no puede tener una definición; Una definición se genera implícitamente, por lo que se puede llamar a la función miembro de evento como si fuera una función miembro normal.

ⓘ Nota

Una clase o estructura con plantilla no puede contener eventos.

Eventos nativos

Los eventos nativos son funciones miembro. El tipo de valor devuelto es normalmente `HRESULT` o `void`, pero puede ser cualquier tipo entero, incluido `enum`. Cuando un evento usa un tipo de valor devuelto entero, se define una condición de error cuando un controlador de eventos devuelve un valor distinto de cero. En este caso, el evento que se genera llama a los demás delegados.

C++

```
// Examples of native C++ events:  
__event void OnDblClick();  
__event HRESULT OnClick(int* b, char* s);
```

Vea [Control de eventos en C++ nativo](#) para ver ejemplos de código.

Eventos COM

Los eventos COM son interfaces. Los parámetros de una función miembro en una interfaz de origen de eventos deben estar *en* los parámetros, pero no se aplican rigurosamente. Es porque un parámetro de *salida* no es útil cuando se realiza la multidifusión. Si usa un parámetro de *salida* se emitirá una advertencia de nivel 1.

El tipo de valor devuelto es normalmente `HRESULT` o `void`, pero puede ser cualquier tipo entero, incluido `enum`. Cuando un evento usa un tipo de valor devuelto entero y un controlador de eventos devuelve un valor distinto de cero, es una condición de error. El evento que se está generando anula las llamadas a los demás delegados. El compilador marca automáticamente una interfaz de origen de eventos como un `source` en el IDL generado.

La palabra clave `_interface` siempre se requiere después de `_event` para un origen de eventos COM.

C++

```
// Example of a COM event:  
__event __interface IEvent1;
```

Consulte el [Control de eventos en COM](#) para obtener un código de muestra.

Eventos administrados

Para obtener información sobre la codificación de eventos en la nueva sintaxis, consulte el [evento](#).

Los eventos administrados son miembros de datos o funciones miembro. Cuando se usa con un evento, el tipo de retorno de un delegado debe cumplir con [Common Language Specification](#). El tipo de valor devuelto del controlador de eventos debe coincidir con el del delegado. Para obtener más información sobre los delegados, consulte [Delegados y eventos](#). Si un evento administrado es un miembro de datos, el tipo debe ser un puntero a un delegado.

En .NET Framework, puede tratar un miembro de datos como si se tratara de un método en sí mismo (es decir, el método `Invoke` de su delegado correspondiente). Para hacerlo, predefina el tipo de delegado para declarar un miembro de datos de eventos administrados. Por el contrario, un método de evento administrado define implícitamente el delegado administrado correspondiente si aún no está definido. Por ejemplo, puede declarar un valor de evento tal como `OnClick` como evento de la manera siguiente:

C++

```
// Examples of managed events:  
__event ClickEventHandler* OnClick; // data member as event  
__event void OnClick(String* s); // method as event
```

Al declarar implícitamente un evento administrado, puede especificar `add` y `remove` a los que se llama cuando se agregan o quitan controladores de eventos. También puede definir la función miembro que llama (genera) el evento desde fuera de la clase.

Ejemplo: eventos nativos

C++

```
// EventHandling_Native_Event.cpp
// compile with: /c
[event_source(native)]
class CSource {
public:
    __event void MyEvent(int nValue);
};
```

Ejemplo: eventos COM

C++

```
// EventHandling_COM_Event.cpp
// compile with: /c
#define _ATL_ATTRIBUTES 1
#include <atlbase.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;
    }
};
```

Consulte también

Palabras clave

Control de eventos

event_source

event_receiver

__hook

__unhook

__raise

palabra clave `_hook`

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

Asocia un método de control a un evento.

ⓘ Nota

Los atributos de eventos en C++ nativo no son compatibles con C++ estándar. No se compilan al especificar el modo de conformidad `/permissive-`.

Sintaxis

C++

```
long __hook(
    &SourceClass::EventMethod,
    source,
    &ReceiverClass::HandlerMethod
    [, receiver = this]
);

long __hook(
    interface,
    source
);
```

Parámetros

`&SourceClass::EventMethod`

Un puntero al método de evento al que se enlaza el método de controlador de eventos:

- Eventos de C++ nativo: `SourceClass` es la clase de origen del evento y `EventMethod` es el evento.
- Eventos COM: `SourceClass` es la interfaz de origen del evento y `EventMethod` es uno de sus métodos.
- Eventos administrados: `SourceClass` es la clase de origen del evento y `EventMethod` es el evento.

`interface`

El nombre de interfaz que se va a enlazar a `receiver`, solo para los receptores de

eventos COM en los que el parámetro `Layout_dependent` del atributo `event_receiver` es `true`.

`source`

Un puntero a una instancia del origen de eventos. Dependiendo del código `type` especificado en `event_receiver`, `source` puede ser uno de estos tipos:

- Un puntero nativo de objeto de origen de eventos.
- Un puntero basado en `IUnknown` (origen COM).
- Un puntero de objeto administrado (para eventos administrados).

`&ReceiverClass::HandlerMethod`

Un puntero al método de controlador de eventos que se a enlazar a un evento. El controlador se especifica como un método de una clase o una referencia a la misma. Si no especifica el nombre de clase, `_hook` asume que la clase es la desde la que se llama.

- Eventos de C++ nativo: `ReceiverClass` es la clase del receptor de eventos y `HandlerMethod` es el controlador.
- Eventos COM: `ReceiverClass` es la interfaz del receptor de eventos y `HandlerMethod` es uno de sus controladores.
- Eventos administrados: `ReceiverClass` es la clase del receptor de eventos y `HandlerMethod` es el controlador.

`receiver`

(Opcional) Un puntero a una instancia de la clase del receptor de eventos. Si no se especifica un receptor, el valor predeterminado es la estructura o la clase del receptor en que se llama a `_hook`.

Uso

Se puede usar en cualquier ámbito de función, incluida main, fuera de la clase del receptor de eventos.

Comentarios

Utilice la función intrínseca `_hook` en un receptor de eventos para asociar o enlazar un método de controlador con un método de evento. Después se llama al controlador

especificado cuando el origen provoca el evento especificado. Puede enlazar varios controladores a un único evento o enlazar varios eventos a un único controlador.

Hay dos formas de `_hook`. Puede usar el primer formulario (de cuatro argumentos) en la mayoría de los casos, concretamente, para los receptores de eventos COM en los que el parámetro `layout_dependent` del atributo `event_receiver` es `false`.

En estos casos no necesita enlazar todos los métodos en una interfaz antes de desencadenar un evento en uno de los métodos. Solo tiene que enlazar el método que controla el evento. Puede usar el segundo formulario (de dos argumentos) de `_hook` solo para un receptor de eventos COM en el que `Layout_dependent = true`.

`_hook` devuelve un valor de tipo long. Un valor devuelto distinto de cero indica que se ha producido un error (los eventos administrados producirán una excepción).

El compilador comprueba la existencia de un evento y que la firma del evento coincida con la firma del delegado.

Puede llamar a `_hook` y `_unhook` fuera del receptor de eventos, excepto los eventos COM.

Una alternativa al uso de `_hook` es utilizar el operador `+=`.

Para obtener información sobre la codificación de eventos administrados en la nueva sintaxis, consulte [event](#).

 **Nota**

Una clase o struct basada en plantilla no puede contener eventos.

Ejemplo

Consulte [Control de eventos en C++ nativo](#) y [Control de eventos en COM](#) para ver ejemplos.

Consulte también

[Palabras clave](#)

[Control de eventos](#)

[event_source](#)

[event_receiver](#)

`_event`

`_unhook`

`_raise`

Palabra clave `__raise`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Resalta el sitio de llamada de un evento.

ⓘ Nota

Los atributos de eventos en C++ nativo no son compatibles con C++ estándar. No se compilan al especificar el modo de conformidad `/permissive-`.

Sintaxis

```
__raise method-declarator ;
```

Comentarios

En código administrado, un evento solo se puede generar desde dentro de la clase donde se define. Para más información, consulte [event](#).

La palabra clave `__raise` hace que se produzca un error si se llama a algún elemento que no es un evento.

ⓘ Nota

Una clase o struct basada en plantilla no puede contener eventos.

Ejemplo

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'
    __raise func2();    // C3745
}
};

int main() {
    E e;
    __raise e.func1();
    __raise e.func1(1); // C3745
    __raise e.func2();    // C3745
}
```

Consulte también

[Palabras clave](#)

[Control de eventos](#)

[_event](#)

[_hook](#)

[_unhook](#)

[Extensiones de componentes de .NET y UWP](#)

Palabra clave `__unhook`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Desvincula un método controlador de un evento.

ⓘ Nota

Los atributos de eventos en C++ nativo no son compatibles con C++ estándar. No se compilan al especificar el modo de conformidad `/permissive-`.

Sintaxis

C++

```
long __unhook(
    &SourceClass::EventMethod,
    source,
    &ReceiverClass::HandlerMethod
    [, receiver = this]
);

long __unhook(
    interface,
    source
);

long __unhook(
    source
);
```

Parámetros

`&SourceClass::EventMethod`

Un puntero al método de evento del que desenlaza el método de controlador de eventos:

- Eventos de C++ nativo: `SourceClass` es la clase de origen del evento y `EventMethod` es el evento.
- Eventos COM: `SourceClass` es la interfaz de origen del evento y `EventMethod` es uno de sus métodos.

- Eventos administrados: `SourceClass` es la clase de origen del evento y `EventMethod` es el evento.

`interface`

El nombre de interfaz que se va a desenlazar de `receiver`, solo para los receptores de eventos COM en los que el parámetro `layout_dependent` del atributo `event_receiver` es `true`.

`source`

Un puntero a una instancia del origen de eventos. Dependiendo del código `type` especificado en `event_receiver`, `source` puede ser uno de estos tipos:

- Un puntero nativo de objeto de origen de eventos.
- Un puntero basado en `IUnknown` (origen COM).
- Un puntero de objeto administrado (para eventos administrados).

`&ReceiverClass::HandlerMethod` Un puntero al método de controlador de eventos que se va a desenlazar de un evento. El controlador se especifica como un método de una clase o una referencia a esta; si no se especifica el nombre de la clase, `__unhook` supone que la clase es aquella en la que se llama.

- Eventos de C++ nativo: `ReceiverClass` es la clase del receptor de eventos y `HandlerMethod` es el controlador.
- Eventos COM: `ReceiverClass` es la interfaz del receptor de eventos y `HandlerMethod` es uno de sus controladores.
- Eventos administrados: `ReceiverClass` es la clase del receptor de eventos y `HandlerMethod` es el controlador.

`receiver`(opcional) Un puntero a una instancia de la clase del receptor de eventos. Si no se especifica un receptor, el valor predeterminado es la estructura o la clase del receptor en que se llama a `__unhook`.

Uso

Se puede usar en cualquier ámbito de función, incluido `main`, fuera de la clase del receptor de eventos.

Comentarios

Use la función intrínseca `_unhook` en un receptor de eventos para desasociar o "desenlazar" un método de controlador de un método de evento.

Hay tres formas de `_unhook`. Puede usar la primera forma (cuatro argumento) en la mayoría de los casos. Puede usar la segunda forma (dos argumentos) de `_unhook` solo para un receptor de eventos COM; de este modo, se desenlaza la interfaz de eventos completa. Puede utilizar la tercera forma (un argumento) para desenlazar todos los delegados del origen especificado.

Un valor devuelto distinto de cero indica que se ha producido un error (los eventos administrados producirán una excepción).

Si se llama a `_unhook` en un evento y un controlador de eventos que ya no están enlazados, no tendrá efecto.

En tiempo de compilación, el compilador comprueba que el evento existe y realiza la comprobación de tipo de parámetros con el controlador especificado.

Puede llamar a `_hook` y `_unhook` fuera del receptor de eventos, excepto los eventos COM.

Una alternativa al uso de `_unhook` es utilizar el operador `-=`.

Para información sobre la creación de código de eventos administrados en la nueva sintaxis, consulte [event](#).

ⓘ Nota

Una clase o struct basada en plantilla no puede contener eventos.

Ejemplo

Consulte [Control de eventos en C++ nativo](#) y [Control de eventos en COM](#) para ver ejemplos.

Consulte también

[Palabras clave](#)

[event_source](#)

[event_receiver](#)

[_event](#)

_hook

_raise

Control de eventos en C++ nativo

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

En el control de eventos de C++ nativo, puede configurar un origen de eventos y un receptor de eventos mediante los atributos `event_source` y `event_receiver`, respectivamente, que especifican `type = native`. Estos atributos permiten a las clases a las que se aplican desencadenar eventos y eventos de control en un contexto nativo, no COM.

ⓘ Nota

Los atributos de eventos en C++ nativo no son compatibles con C++ estándar. No se compilan al especificar el modo de conformidad `/permissive-`.

Declarar eventos

En una clase de origen de eventos, use la palabra clave `_event` en una declaración de método para declarar el método como un evento. Asegúrese de declarar el método, pero no lo defina. Si lo hace, genera un error del compilador, ya que este define el método implícitamente cuando se convierte en un evento. Los eventos nativos pueden ser métodos con cero o más parámetros. El tipo de valor devuelto puede ser `void` o cualquier tipo entero.

Definir controladores de eventos

Puede definir controladores de eventos en una clase receptora de eventos. Los controladores de eventos son métodos con signaturas (tipos de valor devuelto, convenciones de llamada y argumentos) que coinciden con el evento que van a controlar.

Enlazar controladores de eventos a eventos

También en una clase receptora de eventos, se usa la función intrínseca `_hook` para asociar eventos a controladores de eventos y `_unhook` para desasociar eventos de los controladores de eventos. Puede enlazar varios eventos a un controlador de eventos o varios controladores de eventos a un evento.

Desencadenar eventos

Para desencadenar un evento, llame al método declarado como un evento en la clase del origen de eventos. Si se han enlazado controladores al evento, se llamará a los controladores.

Código de evento de C++ nativo

En el ejemplo siguiente se muestra cómo activar un evento en C++ nativo. Para compilar y ejecutar el ejemplo, consulte los comentarios del código. Para compilar el código en el IDE de Visual Studio, compruebe que la opción `/permissive-` está desactivada.

Ejemplo

Código

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);
}
```

Resultados

Output

```
MyHandler2 was called with value 123.  
MyHandler1 was called with value 123.
```

Vea también

[Control de eventos](#)

Control de eventos en COM

Artículo • 03/03/2023 • Tiempo de lectura: 4 minutos

En el control de eventos en COM, puede configurar un origen de eventos y un receptor de eventos usando los atributos `event_source` y `event_receiver`, respectivamente, al especificar `type = com`. Estos atributos insertan código adecuado para interfaces personalizadas, de envío y duales. El código insertado permite que las clases de atributos activen eventos y controlen eventos a través de puntos de conexión COM.

ⓘ Nota

Los atributos de eventos en C++ nativo no son compatibles con C++ estándar. No se compilan al especificar el modo de conformidad `/permissive-`.

Declarar eventos

En una clase de origen del evento, use la palabra clave `_event` en una declaración de interfaz para declarar los métodos de esa interfaz como eventos. Los eventos de esa interfaz se desencadenan cuando se llaman como métodos de interfaz. Los métodos de las interfaces de eventos pueden tener cero o más parámetros (que deben ser todos parámetros *in*). El tipo de valor devuelto puede ser `void` o cualquier tipo entero.

Definición de controladores de eventos

Puede definir controladores de eventos en una clase receptora de eventos. Los controladores de eventos son métodos con signaturas (tipos de valor devuelto, convenciones de llamada y argumentos) que coinciden con el evento que van a controlar. En el caso de los eventos en COM, las convenciones de llamada no tienen que coincidir. Para obtener más información, consulte [Eventos en COM dependientes del diseño](#) a continuación.

Enlazar controladores de eventos a eventos

También en una clase receptora de eventos, se usa la función intrínseca `_hook` para asociar eventos a controladores de eventos y `_unhook` para desasociar eventos de los controladores de eventos. Puede enlazar varios eventos a un controlador de eventos o varios controladores de eventos a un evento.

Nota

Normalmente, hay dos técnicas para permitir que un receptor de eventos COM acceda a las definiciones de interfaz del origen de eventos. La primera, mostrada a continuación, es compartir un archivo de encabezado común. La segunda es usar `#import` con el calificador de importación `embedded_idl`, para que la biblioteca de tipos del origen del evento se escriba en el archivo .tlh con el código generado por el atributo preservado.

Desencadenar eventos

Para desencadenar un evento, simplemente llame a un método de la interfaz declarado con la palabra clave `_event` en la clase del origen del evento. Si se han enlazado controladores al evento, se llamará a los controladores.

Código de eventos en COM

En el ejemplo siguiente se muestra cómo desencadenar un evento en una clase COM. Para compilar y ejecutar el ejemplo, consulte los comentarios del código.

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;
```

Y después el servidor:

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;
    }
};

```

Y después el cliente:

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;
}

```

Output

Output

```

MyHandler1 was called with value 123.
MyHandler2 was called with value 123.

```

Eventos en COM dependientes del diseño

La dependencia de diseño solo es un problema para la programación COM. En el control de eventos nativos y administrados, las signaturas (tipo de valor devuelto, convención de llamada y argumentos) de los controladores deben coincidir con sus eventos, pero los nombres de controlador no tienen que coincidir con sus eventos.

Sin embargo, en el control de eventos en COM, cuando se establece el parámetro `Layout_dependent` de `event_receiver` en `true`, se aplica la coincidencia de nombre y signatura. Los nombres y signaturas de los controladores en el receptor de eventos y en los eventos conectados deben coincidir exactamente.

Cuando `Layout_dependent` se establece en `false`, la convención de llamada y la clase de almacenamiento (virtual, estática, etc.) se pueden combinar y hacer coincidir entre el

método de evento desencadenador y los métodos de enlace (sus delegados). Es un poco más eficiente tener `layout_dependent = true`.

Suponga, por ejemplo, que se define `IEventSource` para que tenga los siguientes métodos:

C++

```
[id(1)] HRESULT MyEvent1([in] int value);
[id(2)] HRESULT MyEvent2([in] int value);
```

Suponga que el origen de eventos tiene el siguiente formato:

C++

```
[coclass, event_source(com)]
class CSource : public IEventSource {
public:
    __event __interface IEvents;

    HRESULT FireEvent() {
        MyEvent1(123);
        MyEvent2(123);
        return S_OK;
    }
};
```

A continuación, en el receptor de eventos, cualquier controlador enlazado a un método en `IEventSource` debe coincidir con el nombre y la firma, de la manera siguiente:

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
}
};
```

Consulte también

[Control de eventos](#)

Modificadores específicos de Microsoft

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

En esta sección se describen las extensiones específicas de Microsoft para C++ en las áreas siguientes:

- [Direccionamiento de base](#), la práctica de usar un puntero como base desde la que se pueden desplazar otros punteros
- [Convenciones de llamadas a función](#)
- Atributos extendidos de clase de almacenamiento declarados con la palabra clave [_declspec](#)
- Palabra clave [_w64](#)

palabras clave específicas de Microsoft

Muchas de las palabras clave específicas de Microsoft se pueden utilizar para modificar declaradores y formar tipos derivados. Para obtener más información sobre los declaradores, vea [Declaradores](#).

| Palabra clave | Significado | ¿Se usa para formar tipos derivados? |
|---------------------------|---|--------------------------------------|
| _based | El nombre que sigue declara un desplazamiento de 32 bits con respecto a la base de 32 bits incluida en la declaración. | Sí |
| _cdecl | El nombre que sigue usa las convenciones de nomenclatura y llamada de C. | Sí |
| _declspec | El nombre que sigue especifica un atributo de clase de almacenamiento específico de Microsoft. | No |
| _fastcall | El nombre que sigue declara una función que usa registros, cuando están disponibles, en lugar de la pila para pasar el argumento. | Sí |
| _restrict | Similar a <code>_declspec(restrict)</code> , pero para usarlo en variables. | No |
| _stdcall | El nombre que sigue especifica una función conforme a la convención de llamada estándar. | Sí |
| _w64 | Marca un tipo de datos como mayor en un compilador de 64 bits. | No |

| Palabra clave | Significado | ¿Se usa para formar tipos derivados? |
|------------------------------|--|--------------------------------------|
| __unaligned | Especifica que un puntero a un tipo u otros datos no esté alineado. | No |
| __vectorcall | El nombre que sigue declara una función que usa registros, incluidos registros de SSE, si están disponibles, en lugar de la pila para el paso de argumentos. | Sí |

Vea también

[Referencia del lenguaje C++](#)

Direccionamiento con base

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Esta sección contiene los siguientes temas:

- [Gramática _based](#)
- [Punteros con base](#)

Consulte también

[Modificadores específicos de Microsoft](#)

__based Gramática

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

El direccionamiento de base es útil cuando se necesita el control preciso del segmento en el que se asignan los objetos (datos basados estáticos y dinámicos).

La única forma de direccionamiento de base aceptable en compilaciones de 32 bits y 64 bits es "basado en un puntero" que define un tipo que contiene un desplazamiento de 32 bits o de 64 bits respecto a una base de 32 bits o de 64 bits, o basado en `void`.

Gramática

based-range-modifier:

`__based(base-expression)`

base-expression:

`based-variable based-abstract-declarator segment-name segment-cast`

based-variable:

`identifier`

based-abstract-declarator:

`abstract-declarator`

base-type:

`type-name`

FIN de Específicos de Microsoft

Consulte también

[Punteros con base](#)

Punteros con base (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

La palabra clave `__based` permite declarar punteros basados en punteros (punteros que son desplazamientos de punteros existentes). La palabra clave `__based` es específica de Microsoft.

Sintaxis

```
type __based( base ) declarator
```

Comentarios

Los punteros basados en direcciones de puntero son la única forma de la palabra clave `__based` válida en compilaciones de 32 y 64 bits. Para el compilador de 32 bits de Microsoft C/C++, un puntero basado es un desplazamiento de 32 bits desde una base de puntero de 32 bits. Se aplica una restricción similar a los entornos de 64 bits, donde un puntero basado es un desplazamiento de 64 bits de la base de 64 bits.

Uno de los usos de los punteros basados en punteros son los identificadores persistentes que contienen punteros. Una lista vinculada formada por punteros basados en un puntero se puede guardar en el disco y recargar en otro lugar de la memoria; los punteros seguirán siendo válidos. Por ejemplo:

C++

```
// based_pointers1.cpp
// compile with: /c
void *vpBuffer;
struct llist_t {
    void __based( vpBuffer ) *vpData;
    struct llist_t __based( vpBuffer ) *llNext;
};
```

Al puntero `vpBuffer` se le asigna la dirección de memoria asignada posteriormente en el programa. La lista vinculada se reubica en relación con el valor de `vpBuffer`.

ⓘ Nota

La persistencia de identificadores que contienen punteros también se consigue mediante **archivos asignados a memoria**.

Cuando se desreferencia un puntero basado, la base se debe especificar explícitamente o se debe conocer implícitamente con la declaración.

Por compatibilidad con versiones anteriores, **_based** es sinónimo de **__based** menos que se especifique la opción del compilador [/Za \(Deshabilitar extensiones de lenguaje\)](#).

Ejemplo

El código siguiente muestra cómo se cambia un puntero basado mediante el cambio de su base.

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
```

Consulte también

[Palabras clave](#)

[alloc_text](#)

Convenciones de llamada

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

El compilador de Visual C/C++ proporciona varias convenciones para llamar a funciones internas y externas. Conocer estos distintos enfoques puede ayudarle a depurar el programa y enlazar el código con rutinas de lenguaje de ensamblado.

Los temas sobre este asunto explican las diferencias entre las convenciones de llamada, cómo se pasan los argumentos y cómo las funciones devuelven valores. También explican las llamadas a función naked, una característica avanzada que permite escribir código de prólogo y epílogo propio.

Para obtener información sobre las convenciones de llamada para los procesadores x64, consulte [Convención de llamada](#).

Temas de esta sección

- [Paso de argumentos y convenciones de nomenclatura \(`__cdecl`, `__stdcall`, `__fastcall`, y otros\)](#)
- [Ejemplo de llamada: Prototipo de función y llamada](#)
- [Uso de llamadas a función naked para escribir código de prólogo/epílogo personalizado](#)
- [Coprocesador de punto flotante y convenciones de llamada](#)
- [Convenciones de llamada obsoletas](#)

Consulte también

[Modificadores específicos de Microsoft](#)

Paso de argumentos y convenciones de nomenclatura

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Los compiladores de Microsoft C++ permiten especificar convenciones para pasar argumentos y valores devueltos entre funciones y llamadores. No todas las convenciones están disponibles en todas las plataformas compatibles y algunas convenciones utilizan implementaciones específicas de la plataforma. En la mayoría de los casos, se omiten palabras clave o modificadores de compilador que especifican una convención no compatible en una plataforma concreta y se usa la convención predeterminada de la plataforma.

En plataformas x86, todos los argumentos se amplían a 32 bits cuando se pasan. Los valores devueltos también se amplían a 32 bits y se devuelven en el registro EAX, salvo las estructuras de 8 bytes, que se devuelven en el par de registros EDX:EAX. Las estructuras de mayor tamaño se devuelven en el registro EAX como punteros a estructuras de devolución ocultas. Los parámetros se insertan en la pila de derecha a izquierda. Las estructuras distintas de POD no se devolverán en registros.

El compilador genera código de prólogo y epílogo para guardar y restaurar los registros ESI, EDI, EBX y EBP, si se usan en la función.

ⓘ Nota

Cuando se devuelve un struct, una unión o una clase desde una función por valor, todas las definiciones del tipo deben ser iguales; de lo contrario, se puede producir un error en el programa en tiempo de ejecución.

Para obtener información sobre la definición de código propio de epílogo y prólogo de la función, vea [Llamadas a funciones naked](#).

Para obtener información sobre las convenciones de llamada predeterminadas en el código destinado a plataformas x64, vea [Convención de llamada x64](#). Para obtener información sobre problemas de convención de llamada en código destinado a plataformas ARM, vea [Problemas comunes de migración de ARM en Visual C++](#).

El compilador de Visual C/C++ admite las siguientes convenciones de llamada.

| Palabra clave | Limpieza de la pila | Paso de parámetros |
|--------------------------|----------------------------|--|
| <code>_cdecl</code> | Autor de llamada | Inserta parámetros en la pila, en orden inverso (de derecha a izquierda) |
| <code>_clrcall</code> | N/D | Carga parámetros en la pila de expresiones CLR en orden (de izquierda a derecha). |
| <code>_stdcall</code> | Destinatario | Inserta parámetros en la pila, en orden inverso (de derecha a izquierda) |
| <code>_fastcall</code> | Destinatario | Almacenado en los registros y luego insertado en la pila |
| <code>_thiscall</code> | Destinatario | Insertado en la pila; puntero <code>this</code> almacenado en ECX |
| <code>_vectorcall</code> | Destinatario | Almacenado en registros y luego insertado en la pila en orden inverso (de derecha a izquierda) |

Para obtener información relacionada, vea [Convenciones de llamada obsoletas](#).

FIN de Específicos de Microsoft

Vea también

[Convenciones de llamada](#)

`_cdecl`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

`_cdecl` es la convención de llamada predeterminada de los programas C y C++. Como el autor de la llamada limpia la pila, puede realizar funciones `vararg`. La convención de llamada `_cdecl` crea archivos ejecutables mayores que `stdcall`, porque requiere que cada llamada a función incluya código de limpieza de la pila. En la lista siguiente se muestra la implementación de esta convención de llamada. El modificador `_cdecl` es específico de Microsoft.

| Elemento | Implementación |
|---|---|
| Orden de paso de argumento | De derecha a izquierda. |
| Responsabilidad de mantenimiento de pila | Al llamar a la función, se extraen los argumentos de la pila. |
| Convención de creación de nombres representativos | El carácter de subrayado (_) precede a los nombres, excepto al exportar funciones <code>_cdecl</code> que usan la vinculación de C. |
| Convención de traducción de mayúsculas y minúsculas | No se lleva a cabo una traducción de mayúsculas y minúsculas. |

ⓘ Nota

Para más información relacionada, consulte [Nombres decorados](#).

Coloque el modificador `_cdecl` delante de una variable o nombre de función. Como las convenciones de llamada y nomenclatura de C son el valor predeterminado, la única vez que se debe usar `_cdecl` en código x86 es cuando se haya especificado la opción del compilador `/Gv` (vectorcall), `/Gz` (stdcall) o `/Gr` (fastcall). La opción del compilador `/Gd` fuerza la convención de llamada `_cdecl`.

En los procesadores ARM y x64, se acepta el uso de `_cdecl`, pero el compilador lo omite normalmente. Por convención en ARM y x64, los argumentos se pasan en registros siempre que es posible y los argumentos subsiguientes se pasan en la pila. En código x64, use `_cdecl` para invalidar la opción del compilador `/Gv` y usar la convención de llamada predeterminada de x64.

En el caso de funciones de clase no estáticas, si la función se define fuera de línea, no es necesario especificar el modificador de convención de llamada en la definición fuera de

línea. Es decir, para los métodos miembro no estáticos de clase, en el momento de la definición se supone la convención de llamada especificada durante la declaración. Dada esta definición de clase:

```
C++  
  
struct CMyClass {  
    void __cdecl mymethod();  
};
```

esto:

```
C++  
  
void CMyClass::mymethod() { return; }
```

equivale a esto:

```
C++  
  
void __cdecl CMyClass::mymethod() { return; }
```

A efectos de compatibilidad con versiones anteriores, `cdecl` y `_cdecl` son sinónimo de `_cdecl`, a menos que se especifique la opción del compilador [/Za \(Deshabilitar extensiones de lenguaje\)](#).

Ejemplo

En el ejemplo siguiente, se le indica al compilador que utilice nombres y convenciones de llamada de C para la función `system`.

```
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, ...);
```

Consulte también

Paso de argumentos y convenciones de nomenclatura

Palabras clave

_clrcall

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

Especifica que solo se puede llamar a una función desde código administrado. Utilice `_clrcall` para todas las funciones virtuales a las que solo se llamará desde código administrado. Sin embargo esta convención de llamada no se puede utilizar para las funciones a las que llamará desde código nativo. El modificador `_clrcall` es específico de Microsoft.

Utilice `_clrcall` para mejorar el rendimiento cuando se llame desde una función administrada a una función administrada virtual o desde una función administrada a una función administrada mediante puntero.

Los puntos de entrada son funciones independientes generadas por el compilador. Si una función tiene puntos de entrada nativos y administrados, uno de ellos será la función real con la implementación de la función. La otra función será una función independiente (un código thunk) que llama a la función real y deja que Common Language Runtime realice PInvoke. Cuando se marca una función como `_clrcall`, se indica que la implementación de la función debe ser MSIL y que la función de punto de entrada nativo no se generará.

Cuando se toma la dirección de una función nativa si no se especifica `_clrcall`, el compilador usa el punto de entrada nativo. `_clrcall` indica que la función es una función administrada y no es necesario realizar la transición de administrada a nativa. En ese caso, el compilador usa el punto de entrada administrado.

Cuando se usa `/clr` (no `/clr:pure` ni `/clr:safe`) y no se usa `_clrcall`, la aceptación de la dirección de una función siempre devuelve la dirección de la función de punto de entrada nativo. Cuando se utiliza `_clrcall`, la función de punto de entrada nativo no se crea, por lo que se obtiene la dirección de la función administrada, no una función de código thunk de punto de entrada. Para más información, consulte [Double Thunking](#). Las opciones del compilador `/clr:pure` y `/clr:safe` están en desuso en Visual Studio 2015 y no se admiten en Visual Studio 2017.

[/clr \(Compilación de Common Language Runtime\)](#) implica que todas las funciones y punteros de función son `_clrcall` y el compilador no permitirá que una función dentro del compilando esté marcada como algo que no sea `_clrcall`. Cuando se utiliza `/clr:pure`, solo se puede especificar `_clrcall` en punteros de función y en declaraciones externas.

Se puede llamar directamente a funciones `_clrcall` desde código de C++ existente que se compiló usando `/clr` siempre que esa función tenga una implementación MSIL. Las

funciones `_clrcall` no se pueden llamar directamente desde funciones con `asm` insertado y a funciones intrínsecas de CPU, por ejemplo, incluso si esas funciones se compilan con `/clr`.

Los punteros de función `_clrcall` solo están diseñados para usarse en el dominio de aplicación en el que se crearon. En lugar de pasar punteros de función `_clrcall` a través de dominios de aplicación, utilice [CrossAppDomainDelegate](#). Para más información, consulte [Dominios de aplicación y Visual C++](#).

Ejemplos

Observe que cuando una función se declare con `_clrcall`, el código se generará cuando sea necesario; por ejemplo, cuando se llame a la función.

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
```

En el ejemplo siguiente se muestra que se puede definir un puntero de función, de modo que se declare que el puntero de función solo se invoque desde código administrado. Esto permite que el compilador llame directamente a la función administrada y evite el punto de entrada nativo (problema de doble código thunk).

C++

```
// clrcall3.cpp
// compile with: /clr
void Test() {
    System::Console::WriteLine("in Test");
}

int main() {
    void (*pTest)() = &Test;
    (*pTest)();

    void (__clrcall *pTest2)() = &Test;
    (*pTest2)();
}
```

Consulte también

[Paso de argumentos y convenciones de nomenclatura](#)
[Palabras clave](#)

__stdcall

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

La convención de llamada `__stdcall` se usa para llamar a funciones de la API de Win32. El destinatario limpia la pila, por lo que el compilador hace funciones `vararg __cdecl`. Las funciones que usan esta convención de llamada requieren un prototipo de función. El modificador `__stdcall` es específico de Microsoft.

Sintaxis

```
| return-type __stdcall function-name[(argument-list)]
```

Comentarios

En la lista siguiente se muestra la implementación de esta convención de llamada.

| Elemento | Implementación |
|---|--|
| Orden de paso de argumento | De derecha a izquierda. |
| Convención para pasar argumentos | Por valor, a menos que se pase un puntero o un tipo de referencia. |
| Responsabilidad de mantenimiento de pila | La función a la que se llama saca sus propios argumentos de la pila. |
| Convención de creación de nombres representativos | Un subrayado (<code>_</code>) precede al nombre. El nombre va seguido del signo (@) seguido del número de bytes (en decimal) en la lista de argumentos. Por consiguiente, la función declarada como <code>int func(int a, double b)</code> se representa de la manera siguiente: <code>_func@12</code> |
| Convención de traducción de mayúsculas y minúsculas | Ninguno |

La opción del compilador [/Gz](#) especifica `__stdcall` para todas las funciones no declaradas explícitamente con otra convención de llamada.

A efectos de compatibilidad con versiones anteriores, `_stdcall` es un sinónimo de `_stdcall` a menos que se especifique la opción del compilador [/Za \(Deshabilitar extensiones de lenguaje\)](#).

Las funciones declaradas con el modificador `_stdcall` devuelven valores del mismo modo que las funciones declaradas con `_cdecl`.

En procesadores ARM y x64, el compilador acepta y omite `_stdcall`; en las arquitecturas ARM y x64, por convención, los argumentos se pasan en registros cuando es posible y los argumentos subsiguientes se pasan en la pila.

En el caso de funciones de clase no estáticas, si la función se define fuera de línea, no es necesario especificar el modificador de convención de llamada en la definición fuera de línea. Es decir, para los métodos miembro no estáticos de clase, en el momento de la definición se supone la convención de llamada especificada durante la declaración.

Dada esta definición de clase,

```
C++  
  
struct CMyClass {  
    void __stdcall mymethod();  
};
```

this

```
C++  
  
void CMyClass::mymethod() { return; }
```

equivale a esto

```
C++  
  
void __stdcall CMyClass::mymethod() { return; }
```

Ejemplo

En el ejemplo siguiente, el uso de `_stdcall` da como resultado que todos los tipos de función `WINAPI` se controlen como una llamada estándar:

```
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, ...);
```

Consulte también

[Paso de argumentos y convenciones de nomenclatura](#)

[Palabras clave](#)

fastcall

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

La convención de llamada **fastcall** especifica que los argumentos de las funciones deben pasarse en registros siempre que sea posible. Esta convención de llamada solo se aplica a la arquitectura x86. En la lista siguiente se muestra la implementación de esta convención de llamada.

| Elemento | Implementación |
|---|--|
| Orden de paso de argumento | Los primeros dos argumentos DWORD o menores que se encuentran en la lista de argumentos de izquierda a derecha se pasan en registros ECX y EDX; el resto de los argumentos se pasan en la pila de derecha a izquierda. |
| Responsabilidad de mantenimiento de pila | Al llamar a la función aparece el argumento de la pila. |
| Convención de creación de nombres representativos | El signo de arroba (@) suele usarse como prefijo para los nombres y el signo de arroba seguido del número de bytes (en formato decimal) en la lista de parámetros se utiliza como sufijo de los nombres. |
| Convención de traducción de mayúsculas y minúsculas | No se lleva a cabo una traducción de mayúsculas y minúsculas. |

ⓘ Nota

Las futuras versiones del compilador pueden utilizar distintos registros para almacenar parámetros.

El uso de la opción del compilador [/Gr](#) hace que cada función del módulo se compile como **fastcall**, salvo que la función se declare con un atributo en conflicto o su nombre sea `main`.

Los compiladores dirigidos a las arquitecturas ARM y x64 aceptan y omiten la palabra clave **fastcall**; en un chip x64, por convención, los primeros cuatro argumentos se pasan en registros cuando sea posible y los argumentos adicionales se pasan en la pila. Para obtener más información, vea [Convención de llamadas x64](#). En un chip ARM, se

puede pasar hasta cuatro argumentos enteros y ocho argumentos de punto flotante en los registros; los argumentos adicionales se pasan en la pila.

En el caso de funciones de clase no estáticas, si la función se define fuera de línea, no es necesario especificar el modificador de convención de llamada en la definición fuera de línea. Es decir, para los métodos miembro no estáticos de clase, en el momento de la definición se supone la convención de llamada especificada durante la declaración.

Dada esta definición de clase:

C++

```
struct CMyClass {  
    void __fastcall mymethod();  
};
```

esto:

C++

```
void CMyClass::mymethod() { return; }
```

equivale a esto:

C++

```
void __fastcall CMyClass::mymethod() { return; }
```

Para la compatibilidad con versiones anteriores, `_fastcall` es sinónimo de `__fastcall`, a menos que se especifique la opción del compilador [/Za \(Deshabilitar extensiones de lenguaje\)](#).

Ejemplo

En el siguiente ejemplo, se pasan argumentos a la función `DeleteAggrWrapper` en los registros:

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, ...);
```

FIN de Específicos de Microsoft

Consulte también

[Paso de argumentos y convenciones de nomenclatura](#)

[Palabras clave](#)

`_thiscall`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

La convención de llamada **específica de Microsoft** `_thiscall` se usa en funciones miembro de clase de C++ en la arquitectura x86. Es la convención de llamada predeterminada que emplean las funciones miembro que no usan argumentos de variable (funciones `vararg`).

En `_thiscall`, el destinatario limpia la pila, lo que es imposible para las funciones `vararg`. Los parámetros se insertan en la pila de derecha a izquierda. El puntero `this` se pasa a través del registro ECX y no en la pila.

En equipos ARM, ARM64 y x64, el compilador acepta y omite a `_thiscall`. Esto se debe a que usan una convención de llamada basada en registros predeterminados.

Un motivo para usar `_thiscall` son las clases cuyas funciones miembro usan `_clrcall` de forma predeterminada. En ese caso, puede usar `_thiscall` para crear funciones miembro individuales que se puedan llamar desde código nativo.

Al compilar con `/clr:pure`, todas las funciones y los punteros a función son `_clrcall` a menos que se especifique lo contrario. Las opciones del compilador `/clr:pure` y `/clr:safe` han quedado en desuso en Visual Studio 2015 y no se admiten en Visual Studio 2017.

Las funciones miembro `vararg` usan la convención de llamada `_cdecl`. Todos los argumentos de función se insertan en la pila, colocándose el puntero `this` en el último lugar de la pila.

Como esta convención de llamada solo se aplica a C++, no tiene ningún esquema de decoración de nombres de C.

Al definir una función miembro de clase no estática fuera de línea, especifique el modificador de convención de llamada solo en la declaración. No es necesario volver a especificarlo en la definición fuera de línea. El compilador usa la convención de llamada especificada durante la declaración en el punto de definición.

Consulte también

[Paso de argumentos y convenciones de nomenclatura](#)

__vectorcall

Artículo • 03/03/2023 • Tiempo de lectura: 14 minutos

Específicos de Microsoft

La convención de llamada **__vectorcall** especifica que los argumentos de las funciones deben pasarse en registros siempre que sea posible. **__vectorcall** usa más registros para argumentos que **_fastcall** o al usar de forma predeterminada la [convención de llamada x64](#). La convención de llamada **__vectorcall** solo se admite en código nativo en procesadores x86 y x64 que incluyen Extensiones SIMD de streaming 2 (SSE2) y versiones posteriores. Use **__vectorcall** para acelerar funciones que pasan varios argumentos de punto flotante o argumentos vectoriales SIMD y llevan a cabo operaciones en las que se usan los argumentos cargados en registros. En la lista siguiente se muestran las características en común con las implementaciones x86 y x64 de **__vectorcall**. Las diferencias se explican más adelante en este artículo.

| Elemento | Implementación |
|--|--|
| Convención de creación de nombres representativos de C | A los nombres de función se les añaden dos "arrobas" (@@) como sufijo, seguidas del número de bytes (en decimal), en la lista de parámetros. |
| Convención de traducción de mayúsculas y minúsculas | No se lleva a cabo la traducción de mayúsculas y minúsculas. |

La opción del compilador [/Gv](#) hace que cada función del módulo se compile como **__vectorcall**, a menos que la función sea una función miembro, se declare con un atributo de convención de llamada incompatible, use una lista de argumentos de variable **vararg** o tenga el nombre **main**.

Puede pasar tres tipos de argumentos por registro en las funciones **__vectorcall**: valores de *tipo entero*, valores de *tipo vectorial* y valores de *agregado vectorial homogéneo* (HVA).

Un tipo entero cumple dos requisitos: se ajusta al tamaño de registro nativo del procesador (por ejemplo, 4 bytes en una máquina x86 o 8 bytes en un equipo x64) y se puede convertir en un entero de longitud de registro y de nuevo sin cambiar su representación de bits. Por ejemplo, es un tipo entero cualquier tipo que se pueda promover a **int** en x86 (**long long** en x64)—por ejemplo, **char** o **short**—o que se pueda convertir a **int** (**long long** en x64) y de nuevo a su tipo original sin cambios. Los tipos de enteros son puntero, referencia y los tipos **struct** o **union** de 4 bytes (8 bytes

en x64) o menos. En las plataformas x64, los tipos `struct` y `union` mayores se pasan por referencia a la memoria asignada por el autor de llamada; en las plataformas x86, se pasan por valor en la pila.

Un tipo vectorial es un tipo de punto flotante (por ejemplo, `float` o `double`) o un tipo de vector SIMD (por ejemplo, `_m128` o `_m256`).

Un tipo HVA es un tipo compuesto de hasta cuatro miembros de datos que tienen tipos vectoriales idénticos. Un tipo HVA tiene el mismo requisito de alineación que el tipo vectorial de sus miembros. Este es un ejemplo de una definición `struct` HVA que contiene tres tipos de vector idénticos y tiene una alineación de 32 bytes:

```
C++  
  
typedef struct {  
    __m256 x;  
    __m256 y;  
    __m256 z;  
} hva3; // 3 element HVA type on __m256
```

Declare las funciones explícitamente con la palabra clave `_vectorcall` en archivos de encabezado para que el código compilado por separado se enlace sin errores. Las funciones deben crear prototipos para usar `_vectorcall` y no pueden usar una `vararg` lista de argumentos de longitud variable.

Una función miembro se puede declarar con el especificador `_vectorcall`. El registro pasa el puntero `this` oculto como el primer argumento de tipo entero.

En equipos ARM, el compilador acepta y omite `_vectorcall`. En el caso de ARM64EC, el compilador no admite y rechaza `_vectorcall`.

En el caso de funciones miembro de clase no estáticas, si la función se define fuera de línea, no es necesario especificar el modificador de convención de llamada en la definición fuera de línea. Es decir, para los miembros de clase no estáticos, se supone que la convención de llamada especificada durante la declaración está en el punto de la definición. Dada esta definición de clase:

```
C++  
  
struct MyClass {  
    void __vectorcall mymethod();  
};
```

esto:

C++

```
void MyClass::mymethod() { return; }
```

equivale a esto:

C++

```
void __vectorcall MyClass::mymethod() { return; }
```

El modificador de convención de llamada `__vectorcall` debe especificarse cuando se crea un puntero a una función `__vectorcall`. En el ejemplo siguiente se crea un `typedef` para un puntero a una función `__vectorcall` que toma cuatro argumentos `double` y devuelve un valor `__m256`:

C++

```
typedef __m256 (__vectorcall * vcfnptr)(double, double, double, double);
```

A efectos de compatibilidad con versiones anteriores, `__vectorcall` es un sinónimo de `__vectorcall1`, a menos que se especifique la opción del compilador [/Za \(Deshabilitar extensiones de lenguaje\)](#).

Convención `__vectorcall` en x64

La convención de llamada `__vectorcall` en x64 amplía la convención de llamada x64 estándar para usar más registros. Los argumentos de tipo entero y los argumentos de tipo vectorial se asignan a registros en función de su posición en la lista de argumentos. Los argumentos de HVA se asignan a los registros vectoriales no usados.

Cuando cualquiera de los cuatro primeros argumentos, por orden, de izquierda a derecha, son argumentos de tipo entero, se pasan en el registro correspondiente a esa posición: RCX, RDX, R8 o R9. Un puntero `this` oculto se trata como el primer argumento de tipo entero. Cuando no se puede pasar un argumento HVA en uno de los cuatro primeros argumentos en los registros disponibles, en su lugar se pasa una referencia a la memoria asignada por el autor de la llamada en el registro de tipo entero correspondiente. Los argumentos de tipo entero después de la cuarta posición de parámetro se pasan en la pila.

Cuando cualquiera de los seis primeros argumentos, por orden, de izquierda a derecha, son argumentos de tipo vectorial, se pasan por valor en los registros vectoriales de SSE

0 a 5, según la posición del argumento. Los tipos de punto flotante y `__m128` se pasan en registros de XMM, y los tipos `__m256` se pasan en registros de YMM. Esto difiere de la convención de llamada x64 estándar, porque los tipos de vector se pasan por valor, no por referencia, y se utilizan registros adicionales. El espacio de pila de propiedad reemplazada asignado para los argumentos de tipo vectorial se fija en 8 bytes, y la opción `/homeparams` no se aplica. Los argumentos de tipo vectorial en las posiciones séptima y posteriores de parámetros se pasan en la pila por referencia a la memoria asignada por el llamador.

Una vez asignados los registros para los argumentos vectoriales, los miembros de datos de los argumentos de HVA se asignan en orden ascendente a los registros vectoriales sin usar XMM0 a XMM5 (o YMM0 a YMM5 para los tipos `__m256`), siempre que haya suficientes registros disponibles para todo el HVA. Si no hay suficientes registros disponibles, el argumento de HVA se pasa por referencia a la memoria asignada por el llamador. El espacio de sombra de pila para un argumento de HVA se fija en 8 bytes con contenido sin definir. Los argumentos de HVA se asignan, por orden, de izquierda a derecha, a los registros de la lista de parámetros, y pueden estar en cualquier posición. Los argumentos de HVA de una de las cuatro primeras posiciones de argumento que no están asignadas a registros vectoriales se pasan por referencia en el registro entero que corresponde a esa posición. Los argumentos de HVA que se pasan por referencia después de la cuarta posición de parámetro se insertan en la pila.

Los resultados de las funciones `__vectorcall` se devuelven por valor en los registros si es posible. Los resultados de tipo entero, incluidos structs o uniones de 8 bytes o menos, se devuelven por valor en RAX. Los resultados de tipo vectorial se devuelven por valor en XMM0 o YMM0, dependiendo del tamaño. Cada elemento de datos de los resultados de HVA se devuelve por el valor en los registros XMM0:XMM3 o YMM0:YMM3, según el tamaño del elemento. Los tipos de resultado que no se adaptan a los registros correspondientes se devuelven por referencia a la memoria asignada por el llamador.

El autor de llamada mantiene la pila en la implementación x64 de `__vectorcall`. El código de prólogo y epílogo del llamador asigna y borra la pila de la función llamada. Los argumentos se insertan en la pila de derecha a izquierda y el espacio de pila de sombra se asigna para los argumentos pasados en registros.

Ejemplos:

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);
}

```

Convención __vectorcall en x86

La convención de llamada `__vectorcall` sigue la convención `__fastcall` para argumentos de tipo entero de 32 bits y usa los registros vectoriales de SSE para los argumentos de HVA y tipo de vector.

Los dos primeros argumentos de tipo entero que se encuentran en la lista de parámetros de izquierda a derecha se colocan en ECX y EDX, respectivamente. Un puntero `this` oculto se trata como el primer argumento de tipo entero y se pasa en ECX. Los seis primeros argumentos de tipo de vector se pasan por valor en los registros vectoriales de SSE del 0 al 5, en los registros de XMM o YMM, dependiendo del tamaño del argumento.

Los seis primeros argumentos de tipo vectorial, por orden, de izquierda a derecha, se pasan por valor en los registros vectoriales de SSE del 0 al 5. Los tipos de punto flotante

y `_m128` se pasan en registros de XMM, y los tipos `_m256` se pasan en registros de YMM. No se asigna ningún espacio de pila de sombra para los argumentos de tipo de vector pasados por registro. Los argumentos de tipo de vector séptimo y posteriores se pasan en la pila por referencia a la memoria asignada por el llamador. La limitación de error del compilador [C2719](#) no se aplica a estos argumentos.

Una vez asignados los registros para los argumentos vectoriales, los miembros de datos de los argumentos de HVA se asignan en orden ascendente a los registros vectoriales sin usar XMM0 a XMM5 (o YMM0 a YMM5 para los tipos `_m256`), siempre que haya suficientes registros disponibles para todo el HVA. Si no hay suficientes registros disponibles, el argumento de HVA se pasa en la pila por referencia a la memoria asignada por el llamador. No se asigna ningún espacio de sombra de pila para un argumento de HVA. Los argumentos de HVA se asignan, por orden, de izquierda a derecha, a los registros de la lista de parámetros, y pueden estar en cualquier posición.

Los resultados de las funciones `_vectorcall` se devuelven por valor en los registros si es posible. Los resultados de tipo entero, incluidos structs o uniones de 4 bytes o menos, se devuelven por valor en EAX. Los structs y uniones de tipo entero de 8 bytes o menos se devuelven por valor en EDX:EAX. Los resultados de tipo vectorial se devuelven por valor en XMM0 o YMM0, dependiendo del tamaño. Cada elemento de datos de los resultados de HVA se devuelve por el valor en los registros XMM0:XMM3 o YMM0:YMM3, según el tamaño del elemento. Otros tipos de resultado se devuelven por referencia a la memoria asignada por el llamador.

La implementación de x86 de `_vectorcall` sigue la convención de los argumentos insertados en la pila de derecha a izquierda por el autor de llamada, y la función llamada borra la pila justo antes de regresar. Solo los argumentos que no se colocan en registros se insertan en la pila.

Ejemplos:

```
C++  
  
// 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);  
}
```

Fin de Específicos de Microsoft

Consulte también

[Paso de argumentos y convenciones de nomenclatura](#)

[Palabras clave](#)

Ejemplo de llamada: Prototipo de función y llamada

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

En el ejemplo siguiente se muestran los resultados de realizar una llamada de función mediante diversas convenciones de llamada.

Este ejemplo está basado en el esqueleto de función siguiente. Reemplace `calltype` por la convención de llamada adecuada.

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);
```

Para obtener más información, vea [Resultados de ejemplo de llamada](#).

FIN de Específicos de Microsoft

Vea también

[Convenciones de llamada](#)

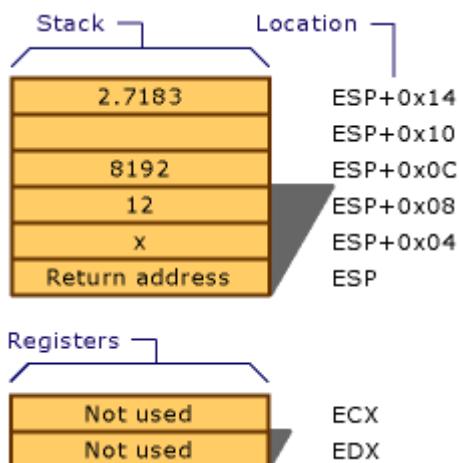
Resultados del ejemplo de llamada

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

`__cdecl`

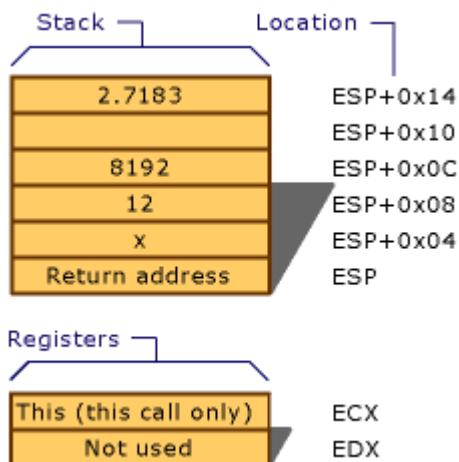
El nombre de función representativo C es `_MyFunc`.



Convención de llamada `__cdecl`

`__stdcall` y `thiscall`

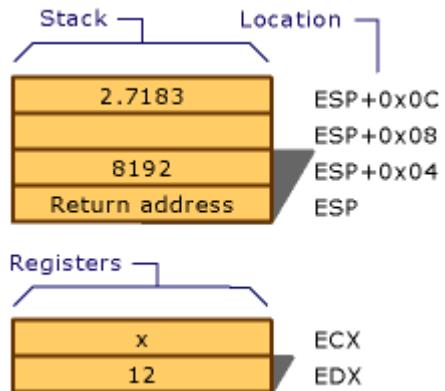
El nombre representativo C (`__stdcall`) es `_MyFunc@20`. El C++ nombre decorado es específico de la implementación.



Las convenciones de llamada de `__stdcall` y `thiscall`

`__fastcall`

El nombre representativo C (`__fastcall`) es `@MyFunc@20`. El C++ nombre decorado es específico de la implementación.



Convención de llamada `__fastcall`

FIN de Específicos de Microsoft

Consulte también

[Ejemplo de llamada: Prototipo de función y llamada](#)

Llamadas de función naked

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Las funciones declaradas con el atributo `naked` se emiten sin código de prólogo o epílogo, lo cual permite escribir secuencias personalizadas de prólogo/epílogo mediante el [ensamblador alineado](#). Las funciones naked se proporcionan como una característica avanzada. Permiten declarar una función que se está llamando desde un contexto que no es C/C++, y crear así diferentes suposiciones sobre dónde están los parámetros o qué registros se conservan. Entre los posibles ejemplos, se encuentran rutinas tales como los controladores de interrupción. Esta característica es especialmente útil para quienes escriben controladores de dispositivos virtuales (VxD).

¿Qué más desea saber?

- [naked](#)
- [Reglas y limitaciones de las funciones naked](#)
- [Consideraciones para escribir código de prólogo/epílogo](#)

FIN de Específicos de Microsoft

Vea también

[Convenciones de llamada](#)

Reglas y limitaciones de las funciones naked

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Las reglas y las limitaciones siguientes se aplican a las funciones naked:

- La instrucción `return` no está permitida.
- Las construcciones de control estructurado de excepciones y control de excepciones de C++ no se permiten porque deben desenredarse a través del marco de la pila.
- Por la misma razón, se prohíbe cualquier forma de `setjmp`.
- Se prohíbe el uso de la función `_alloca`.
- Para asegurarse de que no aparezca ningún código de inicialización para las variables locales antes de la secuencia de prólogo, las variables locales inicializadas no se permiten en el ámbito de la función. En particular, la declaración de objetos de C++ no se permite en el ámbito de la función. Sin embargo, puede haber datos inicializados en un ámbito anidado.
- La optimización del puntero de marco (la opción del compilador /Oy) no se recomienda y se suprime automáticamente en una función naked.
- No se pueden declarar objetos de clase de C++ en el ámbito léxico de la función. Sin embargo, se pueden declarar objetos en un bloque anidado.
- Se omite la palabra clave `naked` al compilar con `/clr`.
- Para las funciones naked de `_fastcall`, siempre que haya una referencia en código de C/C++ para uno de los argumentos del registro, el código de prólogo debe almacenar los valores de ese registro en la ubicación de la pila para esa variable. Por ejemplo:

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
}
}
```

FIN de Específicos de Microsoft

Consulte también

[Llamadas de función naked](#)

Consideraciones para escribir código de prólogo/epílogo

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Antes de escribir sus propias secuencias de código de prólogo y epílogo, es importante que comprenda cómo se diseña el marco de pila. También es útil saber usar el símbolo `_LOCAL_SIZE`.

Diseño del marco de pila

En este ejemplo se muestra el código de prólogo estándar que puede aparecer en una función de 32 bits:

```
push    ebp          ; Save ebp
mov     ebp, esp      ; Set stack frame pointer
sub     esp, localbytes ; Allocate space for locals
push    <registers>   ; Save registers
```

La variable `localbytes` representa el número de bytes necesarios en la pila para las variables locales y la variable `<registers>` es un marcador de posición que representa la lista de registros que se guarden en la pila. Después de insertar los registros, puede colocar cualquier otro dato adecuado en la pila. A continuación, se muestra el código de epílogo correspondiente:

```
pop    <registers>   ; Restore registers
mov     esp, ebp      ; Restore stack pointer
pop     ebp          ; Restore ebp
ret                 ; Return from function
```

La pila siempre va de mayor a menor (de las direcciones de memoria superiores a las inferiores). El puntero base (`ebp`) señala al valor insertado de `ebp`. El área de valores locales comienza en `ebp-4`. Para tener acceso a las variables locales, calcule un desplazamiento de `ebp` restando el valor apropiado a `ebp`.

__LOCAL_SIZE

El compilador proporciona un símbolo, `__LOCAL_SIZE`, para su uso en el bloque de ensamblador alineado del código del prólogo de la función. Este símbolo se utiliza para asignar espacio para variables locales del marco de pila en código de prólogo personalizado.

El compilador determina el valor de `__LOCAL_SIZE`. Su valor es el número total de bytes de todas las variables locales definidas por el usuario y las variables temporales generadas por el compilador. `__LOCAL_SIZE` se puede usar como operando inmediato; no se puede usar en una expresión. No debe cambiar o volver a definir el valor de este símbolo. Por ejemplo:

```
mov      eax, __LOCAL_SIZE          ; Immediate operand--Okay
mov      eax, [ebp - __LOCAL_SIZE]  ; Error
```

En el ejemplo siguiente de una función naked que contiene secuencias personalizadas de prólogo y de epílogo se usa el símbolo `__LOCAL_SIZE` en la secuencia de prólogo:

```
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
    }
}
```

Consulte también

[Llamadas de función naked](#)

Coprocesador de punto flotante y convenciones de llamada

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Si va a escribir rutinas de ensamblado para el coprocesador de punto flotante, debe conservar la palabra de control de punto flotante y limpiar la pila del coprocesador a menos que vaya a devolver un valor `float` o `double` (que debería devolver la función en ST(0)).

Vea también

[Convenciones de llamada](#)

Convenciones de llamada obsoletas

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Las convenciones de llamada `_pascal`, `_fortran` y `_syscall` ya no se admiten. Puede emular su funcionalidad mediante una de las convenciones de llamada admitidas y las opciones del vinculador adecuadas.

`<Windows.h>` admite ahora la macro de WINAPI, que traslada a la convención de llamada adecuada para el destino. Utilice WINAPI donde antes utilizaba PASCAL o `_far _pascal`.

FIN de Específicos de Microsoft

Consulte también

[Paso de argumentos y convenciones de nomenclatura](#)

restrict (C++ AMP)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

El especificador de restricción se puede aplicar a declaraciones de función y lambda. Impone restricciones en el código de la función y en el comportamiento de la función en aplicaciones que utilizan el runtime C++ Accelerated Massive Parallelism (C++ AMP).

ⓘ Nota

Para obtener información sobre la palabra clave `restrict` que forma parte de los atributos de la clase de almacenamiento `_declspec`, vea [restrict](#).

La cláusula `restrict` adopta las formas siguientes:

| Cláusula | Descripción |
|---|---|
| <code>restrict(cpu)</code> | La función puede usar el lenguaje de C++ completo. Solo otras funciones declaradas mediante funciones <code>restrict(cpu)</code> pueden llamar a la función. |
| <code>restrict(amp)</code> | La función solo puede usar el subconjunto del lenguaje C++ que C++ AMP pueda acelerar. |
| Secuencia de <code>restrict(cpu)</code> y <code>restrict(amp)</code> . | La función debe cumplir las limitaciones de <code>restrict(cpu)</code> y <code>restrict(amp)</code> . La función puede ser objeto de llamadas desde funciones declaradas mediante <code>restrict(cpu)</code> , <code>restrict(amp)</code> , <code>restrict(cpu, amp)</code> o <code>restrict(amp, cpu)</code> . La forma <code>restrict(A) restrict(B)</code> puede escribirse como <code>restrict(A,B)</code> . |

Comentarios

La palabra clave `restrict` es una palabra clave contextual. Los especificadores de restricción `cpu` y `amp` no son palabras reservadas. La lista de especificadores no es extensible. Una función que no tiene una cláusula `restrict` es igual que una función con la cláusula `restrict(cpu)`.

Una función que incluye la cláusula `restrict(amp)` tiene las siguientes limitaciones:

- La función puede llamar solo a funciones que tengan la cláusula `restrict(amp)`.
- La función se debe poder insertar.

- La función solo puede declarar variables `int`, `unsigned int`, `float` y `double`, así como las clases y estructuras que contengan solo estos tipos. `bool` también se permite, pero debe ser un tipo alineado de 4 bytes si se usa en un tipo compuesto.
- Las funciones lambda no pueden capturar por referencia, y no pueden capturar punteros.
- Las referencias y los punteros de un solo direccionamiento indirecto se admiten únicamente como variables locales, argumentos de función y tipos de valor devuelto.
- No se permite lo siguiente:
 - La recursividad.
 - Variables declaradas con la palabra clave `volatile`.
 - Funciones virtuales.
 - Punteros a funciones.
 - Punteros a funciones miembro.
 - Punteros en estructuras.
 - Punteros a punteros.
 - Instrucciones `goto`.
 - Instrucciones con etiqueta.
 - Instrucciones `try`, `catch` o `throw`.
 - Variables globales.
 - Variables estáticas. Use `palabra clave tile_static` en su lugar.
 - Conversiones `dynamic_cast`.
 - El operador `typeid`.
 - Declaraciones `asm`.
 - Varargs.

Para obtener una descripción de las limitaciones de función, vea las [restricciones restrict](#).

Ejemplo

En el ejemplo siguiente, se muestra cómo usar la cláusula `restrict(amp)`.

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();
}
```

Consulte también

[C++ AMP \(C++ Accelerated Massive Parallelism\)](#)

tile_static (Palabra clave)

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

La palabra clave **tile_static** se utiliza para declarar una variable a la que pueden tener acceso todos los subprocessos de un mosaico de subprocessos. La duración de la variable comienza cuando la ejecución llega al punto de declaración y termina cuando vuelve la función del kernel. Para obtener más información sobre el uso de mosaicos, consulte [Using Tiles](#).

La palabra clave **tile_static** tiene las limitaciones siguientes:

- Solo se puede utilizar en variables que estén en una función que tenga el modificador `restrict(amp)`.
- No se puede usar en las variables que sean de tipo puntero o referencia.
- Una variable **tile_static** no puede tener un inicializador. Los constructores y destructores predeterminados no se invocan automáticamente.
- El valor de una variable **tile_static** no inicializada es indefinido.
- Si una variable **tile_static** se declara en un gráfico de llamadas cuya raíz se especifica mediante una llamada no en mosaico a `parallel_for_each`, se genera una advertencia y el comportamiento de la variable es indefinido.

Ejemplo

En el ejemplo siguiente, se muestra cómo una variable **tile_static** se puede usar para acumular datos entre varios subprocessos de un mosaico.

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

```

Consulte también

[Modificadores específicos de Microsoft](#)

[Información general sobre C++ AMP](#)

[parallel_for_each \(Función\) \(C++ AMP\)](#)

[Tutorial: Multiplicación de matrices](#)

__declspec

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

Específicos de Microsoft

La sintaxis de atributo extendido para especificar información de clase de almacenamiento utiliza la palabra clave `__declspec`, que especifica que una instancia de un tipo determinado se debe almacenar con un atributo de clase de almacenamiento específico de Microsoft que se muestra a continuación. Otros ejemplos de modificadores de clase de almacenamiento son las palabras clave `static` y `extern`. Sin embargo, estas palabras clave forman parte de la especificación ANSI de los lenguajes C y C++ y, como tales no se incluyen en la sintaxis de atributo extendido. La sintaxis de atributo extendido simplifica y normaliza las extensiones específicas de Microsoft a los lenguajes C y C++.

Gramática

```
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=func-name | ,put=func-name } )
restrict
safebuffers
selectany
spectre(nomitigation)
thread
uuid("ComObjectGUID")
```

El espacio en blanco separa la secuencia de modificador de la declaración. En secciones posteriores aparecen ejemplos.

La gramática de atributos extendidos admite estos atributos de clase de almacenamiento específicos de Microsoft: `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` y `thread`. También admite estos atributos de objetos COM: `property` y `uuid`.

Los atributos de clase de almacenamiento `code_seg`, `dllexport`, `dllimport`, `empty_bases`, `naked`, `noalias`, `nothrow`, `no_sanitize_address`, `property`, `restrict`, `selectany`, `thread` y `uuid` son propiedades solo de la declaración del objeto o función a la que se aplican. El atributo `thread` solo afecta a datos y objetos. Los atributos `naked` y `spectre` afectan solo a las funciones. Los atributos `dllimport` y `dllexport` afectan a funciones, datos y objetos. Los atributos `property`, `selectany` y `uuid` afectan a objetos COM.

A efectos de compatibilidad con versiones anteriores, `_declspec` es un sinónimo de `_declspec` a menos que se especifique la opción del compilador [/Za \(Deshabilitar extensiones de lenguaje\)](#).

Las palabras clave `_declspec` deben colocarse al principio de una declaración simple. El compilador omite, sin advertencia, las palabras clave `_declspec` situadas después de * o & y delante del identificador de variable en una declaración.

Un atributo `_declspec` especificado al comienzo de una declaración de tipos definidos por el usuario se aplica a la variable de ese tipo. Por ejemplo:

C++

```
__declspec(dllexport) class X {} varX;
```

En este caso, el atributo se aplica a `varX`. Un atributo `__declspec` situado después de la palabra clave `class` o `struct` se aplica al tipo definido por el usuario. Por ejemplo:

C++

```
class __declspec(dllexport) X {};
```

En este caso, el atributo se aplica a `X`.

La regla general para utilizar el atributo `__declspec` para las declaraciones simples es la siguiente:

```
decl-specifier-seq init-declarator-list ;
```

El objeto `decl-specifier-seq` debe contener, entre otras cosas, un tipo base (por ejemplo, `int`, `float`, `typedef`o un nombre de clase), una clase de almacenamiento (por ejemplo, `static`, `extern`) o la extensión `__declspec`. `init-declarator-list` debe contener, entre otras cosas, la parte de puntero de declaraciones. Por ejemplo:

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
```

El código siguiente declara una variable local para el subproceso de entero y la inicializa con un valor:

C++

```
// Example of the __declspec keyword
__declspec( thread ) int tls_i = 1;
```

FIN de Específicos de Microsoft

Consulte también

Palabras clave

Atributos extendidos de clase de almacenamiento de C

align (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 8 minutos

En Visual Studio 2015 y versiones posteriores, use el especificador estándar `alignas` de C++11 para la alineación de controles. Para obtener más información, consulte [Alineación](#).

Específicos de Microsoft

Use `__declspec(align(#))` para controlar con precisión la alineación de datos definidos por el usuario (por ejemplo, asignaciones estáticas o datos automáticos en una función).

Sintaxis

`__declspec(align(#))declarator`

Comentarios

Las aplicaciones de escritura que utilizan las últimas instrucciones de procesador presentan nuevas restricciones y problemas. Muchas instrucciones nuevas requieren datos alineados con límites de 16 bytes. La alineación de datos usados con frecuencia con el tamaño de línea de caché del procesador mejora el rendimiento de la memoria caché. Por ejemplo, si define una estructura cuyo tamaño es inferior a 32 bytes, puede que desee alinearla en 32 bytes para asegurarse de que los objetos de ese tipo de estructura se almacenen en caché de forma eficiente.

es el valor de alineación. Las entradas válidas son potencias enteras de dos desde 1 hasta 8192 (bytes), como 2, 4, 8, 16, 32 o 64. `declarator` corresponde a los datos que se declaran como alineados.

Para obtener información sobre cómo devolver un valor de tipo `size_t`, que es el requisito de alineación del tipo, consulte [alignof](#). Para obtener información sobre cómo declarar punteros sin alinear cuando el destino son procesadores de 64 bits, consulte [_unaligned](#).

Puede usar `__declspec(align(#))` al definir `struct`, `union`, o `class` al declarar una variable.

El compilador no garantiza ni intenta conservar el atributo de alineación de datos durante una operación de transformación de datos o copia. Por ejemplo, [memcpy](#)

puede copiar una estructura declarada con `__declspec(align(#))` a cualquier ubicación. Normalmente, los asignadores normales (por ejemplo, `malloc`, `operator new` de C++ y los asignadores win32) devuelven memoria que no está suficientemente alineada para las estructuras `__declspec(align(#))` o matrices de estructuras. Para garantizar que el destino de una operación de transformación de datos o copia está correctamente alineado, use `_aligned_malloc`. O bien, escriba su propio asignador.

No se puede especificar la alineación de los parámetros de la función. Cuando los datos que tiene un atributo de alineación se pasan como un valor en la pila, la alineación se controla con la convención de llamada. Si la alineación de los datos es importante en la función llamada, copie el parámetro en la memoria alineada correctamente antes de su uso.

Sin `__declspec(align(#))`, el compilador suele alinear los datos en los límites naturales basándose en el procesador de destino y el tamaño de los datos, hasta límites de 4 bytes en procesadores de 32 bits y límites de 8 bytes en procesadores de 64 bits. Los datos en clases o estructuras se alinean dentro de la clase o la estructura en el nivel mínimo de su alineación natural y el valor actual de empaquetado (desde `#pragma pack` o la opción `/Zp` del compilador).

Este ejemplo muestra el uso de `__declspec(align(#))`:

C++

```
__declspec(align(32)) struct Str1{
    int a, b, c, d, e;
};
```

Ahora, este tipo tiene un atributo de alineación de 32 bytes. Esto significa que todas las instancias estáticas y automáticas empiezan en un límite de 32 bytes. Otros tipos de estructura declarados con este tipo como miembro conservan el atributo de alineación de este tipo. Es decir, cualquier estructura con `Str1` como elemento tiene un atributo de alineación de al menos 32.

Aquí, `sizeof(struct Str1)` es igual a 32. Esto implica que si se crea una matriz de objetos `Str1`, y la base de la matriz tiene una alineación de 32 bytes, cada miembro de la matriz tiene también una alineación de 32 bytes. Para crear una matriz cuya base esté alineada correctamente en la memoria dinámica, use `_aligned_malloc`. O bien, escriba su propio asignador.

El valor `sizeof` para cualquier estructura es el desplazamiento del miembro final, más el tamaño de ese miembro, redondeado al múltiplo más próximo del mayor valor de

alineación de los miembros o del valor total de alineación de estructura, el que sea mayor.

El compilador utiliza estas reglas para la alineación de la estructura:

- A menos que se reemplace con `__declspec(align(#))`, la alineación de un miembro de estructura escalar es el mínimo de su tamaño y empaquetado actual.
- A menos que se reemplace con `__declspec(align(#))`, la alineación de una estructura es el máximo de las alineaciones individuales de sus miembros.
- Un miembro de estructura se coloca en un desplazamiento desde el inicio de su estructura primaria que es el múltiplo más pequeño de su alineación mayor o igual que el desplazamiento del final del miembro anterior.
- El tamaño de una estructura es el múltiplo más pequeño de su alineación, que es mayor o igual que el desplazamiento del final de su último miembro.

`__declspec(align(#))` solo puede aumentar restricciones de alineación.

Para más información, consulte:

- [align Ejemplos](#)
- [Definición de nuevos tipos con `__declspec\(align\(#\)\)`](#)
- [Alinear datos en almacenamiento local para el subprocesso](#)
- [Cómo align funciona con el empaquetado de datos](#)
- [Ejemplos de alineación de estructuras x64](#)

align Ejemplos

En los ejemplos siguientes se muestra cómo `__declspec(align(#))` afecta al tamaño y la alineación de estructuras de datos. En los ejemplos se suponen las definiciones siguientes:

C++

```
#define CACHE_LINE 32
#define CACHE_ALIGN __declspec(align(CACHE_LINE))
```

En este ejemplo, la `s1` estructura se define mediante el uso de `__declspec(align(32))`. Todos los usos de `s1` para una definición de variable o en otro tipo de declaraciones tienen una alineación de 32 bytes. `sizeof(struct s1)` devuelve 32 y `s1` tiene 16 bytes de relleno a continuación de los 16 bytes necesarios para contener los cuatro enteros.

Cada miembro `int` requiere una alineación de 4 bytes, pero la alineación de la propia estructura se declara de 32. Por lo tanto, la alineación total es 32.

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
```

En este ejemplo, `sizeof(struct S2)` devuelve 16, que es exactamente la suma de los tamaños de miembro, porque es un múltiplo del mayor requisito de alineación (un múltiplo de 8).

C++

```
__declspec(align(8)) struct S2 {
    int a, b, c, d;
};
```

En el ejemplo siguiente, `sizeof(struct S3)` devuelve 64.

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
};
```

En este ejemplo, observe que `a` tiene la alineación de su tipo natural, en este caso, 4 bytes. Sin embargo, `s1` debe tener una alineación de 32 bytes. 28 bytes de relleno siguen a `a`, de modo que `s1` empieza en la posición de desplazamiento 32. `s4` hereda después el requisito de alineación de `s1`, porque es el requisito de alineación mayor de la estructura. `sizeof(struct S4)` devuelve 64.

C++

```
struct S4 {
    int a;
    // 28 bytes padding
    struct S1 s1; // S4 inherits cache alignment requirement of S1
};
```

Las tres declaraciones de variable siguientes también utilizan `__declspec(align(#))`. En cada caso, la variable debe tener una alineación de 32 bytes. En la matriz, la dirección base de la matriz, no cada miembro de la matriz, tiene una alineación de 32 bytes. El valor `sizeof` para cada miembro de la matriz no se ve afectado por el uso de `__declspec(align(#))`.

C++

```
CACHE_ALIGN int i;
CACHE_ALIGN int array[128];
CACHE_ALIGN struct s2 s;
```

Para alinear cada miembro de una matriz, se debe utilizar código como el siguiente:

C++

```
typedef CACHE_ALIGN struct { int a; } S5;
S5 array[10];
```

En este ejemplo, observe que la alineación de la propia estructura y la alineación del primer elemento tienen el mismo efecto:

C++

```
CACHE_ALIGN struct S6 {
    int a;
    int b;
};

struct S7 {
    CACHE_ALIGN int a;
    int b;
};
```

`S6` y `S7` tienen características de alineación, asignación y tamaño idénticas.

En este ejemplo, la alineación de las direcciones iniciales de `a`, `b`, `c` y `d` son 4, 1, 4 y 1, respectivamente.

C++

```
void fn() {
    int a;
    char b;
    long c;
```

```
char d[10]
}
```

La alineación cuando la memoria se asignó en el montón depende de qué función de asignación se invoque. Por ejemplo, si utiliza `malloc`, el resultado depende del tamaño de operando. Si $arg \geq 8$, la memoria devuelta tiene una alineación de 8 bytes. Si $arg < 8$, la alineación de la memoria devuelta es la primera potencia de 2 menor que arg . Por ejemplo, si usa `malloc(7)`, la alineación es de 4 bytes.

Definición de nuevos tipos con `__declspec(align(#))`

Puede definir un tipo con una característica de alineación.

Por ejemplo, puede definir un `struct` con un valor de alineación de la forma siguiente:

C++

```
struct aType {int a; int b;};
typedef __declspec(align(32)) struct aType bType;
```

Ahora, `aType` y `bType` tienen el mismo tamaño (8 bytes) pero las variables de tipo `bType` tienen una alineación de 32 bytes.

Alinear datos en el almacenamiento local para el subproceso

El almacenamiento local de subprocessos estáticos (TLS) creado con el atributo `__declspec(thread)` y colocado en la sección de TLS en la imagen funciona para la alineación exactamente igual que los datos estáticos normales. Para crear datos TLS, el sistema operativo asigna a la memoria el tamaño de la sección de TLS y respetando el atributo de la alineación de la sección de TLS.

En este ejemplo se muestran diversas maneras de colocar datos alineados en el almacenamiento local para el subproceso.

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;

```

Cómo `align` funciona con el empaquetado de datos

La opción del compilador `/zp` y la pragma de `pack` tienen el efecto de empaquetar datos para los miembros de estructura y de unión. En este ejemplo se muestra cómo `/zp` y `__declspec(align(#))` trabajan juntos:

C++

```

struct S {
    char a;
    short b;
    double c;
    CACHE_ALIGN double d;
    char e;
    double f;
};

```

En la tabla siguiente se muestra el desplazamiento de cada miembro bajo diversos valores de `/zp` (o `#pragma pack`), al mostrar cómo interactúan ambos.

| Variable | <code>/Zp1</code> | <code>/Zp2</code> | <code>/Zp4</code> | <code>/Zp8</code> |
|------------------------|-------------------|-------------------|-------------------|-------------------|
| 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 |
| <code>sizeof(S)</code> | 64 | 64 | 64 | 64 |

Para obtener más información, consulte [/Zp \(Alineación de miembros de estructura\)](#).

El desplazamiento de un objeto se basa en el desplazamiento del objeto anterior y el valor actual de empaquetado, a menos que el objeto tenga un atributo `__declspec(align(#))`, en cuyo caso la alineación se basa en el desplazamiento del objeto anterior y el valor `__declspec(align(#))` para el objeto.

FIN de Específicos de Microsoft

Vea también

[__declspec](#)

[Información general sobre las convenciones ABI de ARM](#)

[Convenciones de software x64](#)

allocate

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

El especificador de declaración `allocate` designa el segmento de datos en el que se asignará el elemento de datos.

Sintaxis

```
| __declspec(allocate("segname")) declarator
```

Comentarios

El nombre *segname* se debe declarar con una de las instrucciones pragma siguientes:

- `code_seg`
- `const_seg`
- `data_seg`
- `init_seg`
- `section`

Ejemplo

C++

```
// allocate.cpp
#pragma section("mycode", read)
__declspec(allocate("mycode")) int i = 0;

int main() {
}
```

FIN de Específicos de Microsoft

Consulte también

_declspec

Palabras clave

allocator

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

El especificador de declaración `allocator` se puede aplicar a las funciones de asignación de memoria personalizadas para que las asignaciones sean visibles a través del seguimiento de eventos para Windows (ETW).

Sintaxis

```
| __declspec(allocator)
```

Comentarios

El generador de perfiles de memoria nativa en Visual Studio funciona mediante la recopilación de datos de asignación de eventos de ETW que se emiten en tiempo de ejecución. Los asignadores de CRT y Windows SDK se han anotado en el nivel de origen para que se puedan capturar los datos de asignación. Si escribe sus propios asignadores, las funciones que devuelven un puntero a la memoria de montón recientemente asignada se pueden decorar con `__declspec(allocator)`, tal como se muestra en este ejemplo para `myMalloc`:

C++

```
__declspec(allocator) void* myMalloc(size_t size)
```

Para obtener más información, consulte [Medición del uso de memoria en Visual Studio](#) y [Eventos de montón ETW nativos personalizados](#).

FIN de Específicos de Microsoft

appdomain

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

Especifica que cada dominio de aplicación de la aplicación administrada debe tener una copia propia de una variable global determinada o de una variable miembro static.

Consulte [Dominios de aplicación y Visual C++](#) para obtener más información.

Cada dominio de aplicación tiene su propia copia de una variable por appdomain. Un constructor de una variable appdomain se ejecuta cuando se carga un ensamblado en un dominio de aplicación, y se ejecuta el destructor cuando se descarga el dominio de aplicación.

Si desea que todos los dominios de aplicación de un proceso de Common Language Runtime compartan una variable global, utilice el modificador `__declspec(process)`.

`__declspec(process)` está en vigor de forma predeterminada bajo `/clr`. Las opciones del compilador `/clr:pure` y `/clr:safe` han quedado en desuso en Visual Studio 2015 y no se admiten en Visual Studio 2017.

`__declspec(appdomain)` es válido solo cuando se usa una de las opciones del compilador `/clr`. Solo una variable global, una variable miembro estática o una variable local estática se pueden marcar con `__declspec(appdomain)`. Es un error aplicar `__declspec(appdomain)` a miembros estáticos de tipos administrados, porque siempre tienen este comportamiento.

Usar `__declspec(appdomain)` es similar a usar el [almacenamiento local para el subproceso \(TLS\)](#). Los subprocesos tienen su propio almacenamiento, como los dominios de aplicación. Al usar `__declspec(appdomain)`, se garantiza que la variable global tiene su propio almacenamiento en cada dominio de aplicación creado para esta aplicación.

Hay limitaciones para la combinación del uso de variables por proceso y por dominio de aplicación; consulte [proceso](#) para obtener más información.

Por ejemplo, en el inicio del programa, primero se inicializan todas las variables por proceso y, después, todas las variables por appdomain. Por consiguiente, cuando se inicializa una variable por proceso, esta no puede depender del valor de cualquier variable de dominio por aplicación. No se recomienda combinar el uso (asignación) de variables por appdomain y por proceso.

Para obtener información sobre cómo llamar a una función en un dominio de aplicación concreto, consulte [Función call_in_appdomain](#).

Ejemplo

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

```

Consulte también

[_declspec](#)

[Palabras clave](#)

`__declspec(code_seg)`

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

Específicos de Microsoft

El `code_seg` atributo de declaración asigna un segmento de texto ejecutable al archivo en el `.obj` que se almacena el código de objeto de las funciones miembro de la función o clase.

Sintaxis

```
__declspec(code_seg(" segname ")) declarator
```

Comentarios

El atributo `__declspec(code_seg(...))` permite colocar código en segmentos con nombre diferentes que se pueden paginar o bloquear en memoria individualmente. Este atributo se puede usar para controlar la posición de las plantillas con instancia y del código generado por el compilador.

Un *segmento* es un bloque con nombre de datos en un `.obj` archivo que se carga en la memoria como una unidad. Un *segmento de texto* es un segmento que contiene código ejecutable. El término *sección* se suele usar indistintamente con el término segmento.

El código objeto que se genera cuando se define `declarator` se coloca en el segmento de texto especificado por `segname`, que es un literal de cadena de caracteres estrechos. El nombre `segname` no tiene que especificarse en una [pragma de sección](#) para poder usarse en una declaración. De forma predeterminada, cuando no se especifica ningún `code_seg`, el código de objeto se coloca en un segmento denominado `.text`. Un `code_seg` atributo invalida cualquier directiva `#pragma code_seg` existente. Un `code_seg` atributo aplicado a una función miembro invalida cualquier `code_seg` atributo aplicado a la clase envolvente.

Si una entidad tiene un `code_seg` atributo, todas las declaraciones y definiciones de la misma entidad deben tener atributos idénticos `code_seg`. Si una clase base tiene un `code_seg` atributo, las clases derivadas deben tener el mismo atributo.

Cuando se aplica un `code_seg` atributo a una función de ámbito de espacio de nombres o a una función miembro, el código de objeto de esa función se coloca en el segmento

de texto especificado. Cuando este atributo se aplica a una clase, todas las funciones miembro de la clase y las clases anidadas (incluidas las funciones miembro especiales generadas por el compilador) se colocan en el segmento especificado. Las clases definidas localmente (por ejemplo, las clases definidas en un cuerpo de función miembro) no heredan el `code_seg` atributo del ámbito envolvente.

Cuando se aplica un `code_seg` atributo a una plantilla de clase o a una plantilla de función, todas las especializaciones implícitas de la plantilla se colocan en el segmento especificado. Las especializaciones explícitas o parciales no heredan el `code_seg` atributo de la plantilla principal. Puede especificar el mismo atributo o otro `code_seg` en la especialización. Un `code_seg` atributo no se puede aplicar a una creación de instancias de plantilla explícita.

De forma predeterminada, el código generado por el compilador, como una función miembro especial, se coloca en el `.text` segmento. La `#pragma code_seg` directiva no invalida este valor predeterminado. Use el `code_seg` atributo en la clase, la plantilla de clase o la plantilla de función para controlar dónde se coloca el código generado por el compilador.

Las expresiones lambda heredan `code_seg` los atributos de su ámbito envolvente. Para especificar un segmento para una expresión lambda, aplique un `code_seg` atributo después de la cláusula `parameter-declaration` y antes de cualquier especificación mutable o de excepción, cualquier especificación final de tipo de valor devuelto y el cuerpo lambda. Para obtener más información, vea [Sintaxis de la expresión lambda](#). En este ejemplo se define una expresión lambda en un segmento denominado `PagedMem`:

C++

```
auto Sqr = [](int t) __declspec(code_seg("PagedMem")) -> int { return t*t;  
};
```

Tenga cuidado cuando coloque determinadas funciones miembro, especialmente funciones miembro virtuales, en segmentos diferentes. Supongamos que define una función virtual en una clase derivada que reside en un segmento paginado cuando el método de clase base reside en un segmento no paginado. Otros métodos de clase base o código de usuario pueden suponer que invocar el método virtual no desencadenará un error de página.

Ejemplo

En este ejemplo se muestra cómo un atributo controla la `code_seg` selección de ubicación del segmento cuando se usa la especialización implícita y explícita de plantillas:

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);
}
```

FIN de Específicos de Microsoft

Consulte también

[_declspec](#)

[Palabras clave](#)

en desuso (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Este tema trata sobre la declaración `declspec` específica de Microsoft en desuso. Para obtener información sobre el atributo C++14 `[[deprecated]]` y guía sobre cuándo usar ese atributo frente a la `declspec` o pragma específica de Microsoft, vea [Atributos estándar de C++](#).

Con las excepciones que se indican a continuación, la declaración `deprecated` proporciona la misma funcionalidad que la directiva pragma [en desuso](#):

- La declaración `deprecated` en desuso permite especificar formas determinadas de sobrecargas de función como en desuso, mientras que la forma pragma se aplica a todas las formas sobrecargadas de un nombre de función.
- La declaración `deprecated` permite especificar un mensaje que se mostrará en el tiempo de compilación. El texto del mensaje puede proceder de una macro.
- Las macros solo se pueden marcar como en desuso con la pragma `deprecated`.

Si el compilador encuentra el uso de un identificador en desuso o el atributo `[[deprecated]]` estándar, se produce una advertencia [C4996](#).

Ejemplos

En el ejemplo siguiente se muestra cómo marcar funciones como desusadas y cómo especificar un mensaje que se mostrará, en tiempo de compilación, cuando se use la función desusada.

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  
}
```

En el ejemplo siguiente se muestra cómo marcar clases como desusadas y cómo especificar un mensaje que se mostrará, en tiempo de compilación, cuando se use la clase desusada.

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
}
```

Consulte también

[__declspec](#)

[Palabras clave](#)

dllexport, dllimport

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Los atributos de clase de almacenamiento `dllexport` y `dllimport` son extensiones específicas de Microsoft para los lenguajes C y C++. Se pueden utilizar para exportar e importar funciones, datos y objetos a o de una DLL.

Sintaxis

```
__declspec( dllimport ) declarator  
__declspec( dllexport ) declarator
```

Comentarios

Estos atributos definen explícitamente la interfaz de la DLL para el cliente, que puede ser el archivo ejecutable u otra DLL. Declarar funciones como `dllexport` elimina la necesidad de un archivo de definición de módulos (.def), al menos en relación con la especificación de funciones exportadas. El atributo `dllexport` reemplaza la palabra clave `__export`.

Si una clase está marcada con `__declspec(dllexport)`, cualquier especialización de las plantillas de clase en la jerarquía de clases se marca implícitamente como `__declspec(dllexport)`. Esto significa que se crean explícitamente instancias de las plantillas de clase y que los miembros de la clase se deben definir.

`dllexport` de una función expone la función con su nombre decorado, a veces conocido como "mangling de nombres". Para las funciones de C++, el nombre decorado incluye caracteres adicionales que codifican el tipo y la información de parámetros. Las funciones de C o las funciones declaradas como `extern "C"` incluyen la decoración específica de la plataforma que se basa en la convención de llamada. No se aplica ninguna decoración de nombres a las funciones exportadas de C ni a las funciones `extern "C"` de C++ que usan la convención de llamada `__cdecl`. Para más información sobre la decoración de nombres en el código de C o C++, vea [Nombres representativos](#).

Para exportar un nombre no representativo, puede vincular mediante un archivo de definición de módulo (.def) que define el nombre no representativo de una sección `EXPORTS`. Para más información, consulte [EXPORTS](#). Otra forma de exportar un nombre

no representativo es usar una directiva `#pragma comment(linker, "/export:alias=decorated_name")` en el código fuente.

Cuando declare `__declspec(dllexport)` o `__declspec(dllimport)`, debe utilizar la [sintaxis de atributo extendida](#) y la palabra clave `__declspec`.

Ejemplo

C++

```
// Example of the __declspec(dllexport) and __declspec(dllimport) class attributes
__declspec(dllimport) int i;
__declspec(dllexport) void func();
```

Opcionalmente, para que el código sea más legible, puede utilizar definiciones de macro:

C++

```
#define DllImport    __declspec(dllimport)
#define DllExport    __declspec(dllexport)

DllExport void func();
DllExport int i = 10;
DllImport int j;
DllExport int n;
```

Para más información, consulte:

- [Definiciones y declaraciones](#)
- [Definición de funciones C++ insertadas con __declspec\(dllexport\) y __declspec\(dllimport\)](#)
- [Reglas generales y limitaciones](#)
- [Uso __declspec\(dllexport\) y __declspec\(dllexport\) en clases de C++](#)

FIN de Específicos de Microsoft

Consulte también

[__declspec](#)

[Palabras clave](#)

Definiciones y declaraciones (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

La interfaz DLL hace referencia a todos los elementos (funciones y datos) que se sabe que se van a exportar mediante algún programa del sistema, es decir, todos los elementos declarados como `dllimport` o `dllexport`. Todas las declaraciones incluidas en la interfaz DLL deben especificar el atributo `dllimport` o `dllexport`. Sin embargo, la definición debe especificar el atributo `dllexport`. Por ejemplo, la definición de función siguiente genera un error del compilador:

```
__declspec( dllimport ) int func() {    // Error; dllimport
                                         // prohibited on definition.
    return 1;
}
```

Este código también genera un error:

```
__declspec( dllimport ) int i = 10; // Error; this is a definition.
```

Sin embargo, esta es la sintaxis correcta:

```
__declspec( dllexport ) int i = 10; // Okay--export definition
```

El uso de `dllexport` implica una definición, mientras que `dllimport` implica una declaración. Debe utilizar la palabra clave `extern` con `dllexport` para forzar una declaración; en caso contrario, se asume una definición. Por tanto, los siguientes ejemplos son correctos:

```
#define DllImport __declspec( dllimport )
#define DllExport __declspec( dllexport )

extern DllExport int k; // These are both correct and imply a
                      // declaration.
                      // declaration.
```

Los ejemplos siguientes aclaran los anteriores:

```
static __declspec( dllexport ) int l; // Error; not declared extern.

void func() {
    static __declspec( dllimport ) int s; // Error; not declared
                                         // extern.
    __declspec( dllimport ) 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.
}
```

FIN de Específicos de Microsoft

Consulte también

[dllexport, dllimport](#)

Definir funciones insertadas de C++ con `dllexport` y `dllimport`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Puede definir como alineada una función con el atributo `dllexport`. En este caso, siempre se crean instancias de la función, y siempre se exporta, independientemente de si un módulo del programa hace referencia o no a la función. Se supone que otro programa importa la función.

También puede definir como insertada una función declarada con el atributo `dllimport`. En este caso, la función se puede expandir (según las especificaciones de /Ob), pero nunca se pueden crear instancias de ella. En concreto, si se toma la dirección de una función alineada importada, se devuelve la dirección de la función que reside en la DLL. Este comportamiento es el mismo que cuando se toma la dirección de una función importada no alineada.

Estas reglas se aplican a las funciones insertadas cuyas definiciones aparecen dentro de una definición de clase. Además, los datos y cadenas locales estáticos en funciones insertadas mantienen las mismas identidades entre la DLL y el cliente que en un solo programa (es decir, un archivo ejecutable sin interfaz DLL).

Tome precauciones cuando proporcione funciones insertadas importadas. Por ejemplo, si actualiza la DLL, no suponga que el cliente utilizará la versión modificada de la DLL. Para asegurarse de que se carga la versión correcta, recompile el cliente de DLL también.

FIN de Específicos de Microsoft

Consulte también

[dllexport, dllimport](#)

Reglas generales y limitaciones

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

Específicos de Microsoft

- Si declara una función o un objeto sin el atributo `dllimport` o `dllexport`, la función o el objeto no se consideran parte de la interfaz DLL. Por consiguiente, la definición de la función o el objeto debe estar presente en ese módulo o en otro módulo del mismo programa. Para que una función o un objeto formen parte de la interfaz DLL, debe declarar la definición de la función o del objeto en el otro módulo como `dllexport`. De los contrario, se genera un error del vinculador.

Si declara una función o un objeto con el atributo `dllexport`, su definición debe aparecer en algún módulo del mismo programa. De los contrario, se genera un error del vinculador.

- Si un solo módulo del programa contiene declaraciones `dllimport` y `dllexport` para la misma función u objeto, el atributo `dllexport` tiene prioridad sobre el atributo `dllimport`. Sin embargo, se genera una advertencia del compilador. Por ejemplo:

C++

```
__declspec( dllimport ) int i;
__declspec( dllexport ) int i;    // Warning; inconsistent;
                                // dllexport takes precedence.
```

- En C++, puede inicializar un puntero de datos local estático o declarado globalmente o con la dirección de un objeto de datos declarado con el atributo `dllimport`, lo que genera un error en C. Además, puede inicializar un puntero de función local estático con la dirección de una función declarada con el atributo `dllimport`. En C, esta asignación establece el puntero en la dirección del código thunk de importación de DLL (un código auxiliar que transfiere el control a la función) en lugar de en la dirección de la función. En C++, establece el puntero en la dirección de la función. Por ejemplo:

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++
}
```

Aun así, como un programa que incluya el atributo `__declspec(dllexport)` en la declaración de un objeto debe proporcionar la definición de ese objeto en alguna parte del programa, puede inicializar un puntero de función estático global o local con la dirección de una función `__declspec(dllexport)`. De igual forma, puede inicializar un puntero de datos estático global o local con la dirección de un objeto de datos `__declspec(dllexport)`.

Por ejemplo, el código siguiente no genera errores en C ni en C++:

```
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
}
```

- Si se aplica `__declspec(dllexport)` a una clase normal que tiene una clase base que no está marcada como `__declspec(dllexport)`, el compilador generará la advertencia C4275.

El compilador genera la misma advertencia si la clase base es una especialización de una plantilla de clase. Para solucionar este problema, marque la clase base con `__declspec(dllexport)`. El problema con una especialización de una plantilla de clase es dónde colocar `__declspec(dllexport)`; no se permite marcar la plantilla de clase. En lugar de ello, cree explícitamente una instancia de la plantilla de clase y marque esta instancia explícita con `__declspec(dllexport)`. Por ejemplo:

```
C++

template class __declspec(dllexport) B<int>;
class __declspec(dllexport) D : public B<int> {
// ...
```

Esta solución no funciona si el argumento de la plantilla es la clase de la que se deriva. Por ejemplo:

C++

```
class __declspec(dllexport) D : public B<D> {  
// ...
```

Como este es un patrón común con plantillas, el compilador cambió la semántica de `dllexport` cuando se aplica a una clase que tiene una o más clases base y cuando una o más de las clases base es una especialización de una plantilla de clase. En este caso, el compilador aplica implícitamente `dllexport` a las especializaciones de plantillas de clase. Puede hacer lo siguiente y no recibir una advertencia:

C++

```
class __declspec(dllexport) D : public B<D> {  
// ...
```

FIN de Específicos de Microsoft

Consulte también

[dllexport, dllimport](#)

Utilizar `dllimport` y `dllexport` en las clases de C++

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

Específicos de Microsoft

Se pueden declarar las clases de C++ con el atributo `dllimport` o `dllexport`. Estas formas implican que se importa o exporta la clase completa. Las clases exportadas de esta forma se denominan clases exportables.

En el ejemplo siguiente se define una clase exportable. Se exportan todas sus funciones miembro y todos sus datos estáticos:

C++

```
#define DllExport __declspec( dllexport )

class DllExport C {
    int i;
    virtual int func( void ) { return 1; }
};
```

Observe que está prohibido el uso explícito de los atributos `dllimport` y `dllexport` en los miembros de una clase exportable.

`dllexport` (Clases)

Cuando se declara una clase `dllexport`, se exportan todas sus funciones miembro y todos sus miembros de datos estáticos. Debe proporcionar las definiciones de todos esos miembros en el mismo programa. De lo contrario, se genera un error del vinculador. La única excepción a esta regla afecta a las funciones virtuales puras, para las que no es necesario proporcionar definiciones explícitas. Sin embargo, dado que a un destructor de una clase abstracta siempre lo invoca el destructor de la clase base, los destructores virtuales puros deben proporcionar siempre una definición. Observe que estas reglas son las mismas para las clases no exportables.

Si exporta datos de tipo de clase o funciones que devuelven clases, asegúrese de exportar la clase.

`dllimport` (Clases)

Cuando se declara una clase `dllimport`, se importan todas sus funciones miembro y todos sus miembros de datos estáticos. A diferencia del comportamiento de `dllimport` y `dllexport` en tipos que no son de clase, los miembros de datos estáticos no pueden especificar una definición en el mismo programa en el que se define una clase `dllimport`.

Herencia y clases exportables

Todas las clases base de una clase exportable deben ser exportables. De no ser así, se genera una advertencia del compilador. Además, todos los miembros accesibles que también son clases deben ser exportables. Esta regla permite que una clase `dllexport` herede de una clase `dllimport` y una clase `dllimport` herede de una clase `dllexport` (aunque esto último no se recomienda). Como norma, todo lo que es accesible al cliente de la DLL (de acuerdo con las reglas de acceso de C++) debe formar parte de la interfaz exportable. En esto se incluye a los miembros de datos privados a que se hace referencia en funciones insertadas.

Importación/exportación selectiva de miembros

Debido a que las funciones miembro y los datos estáticos dentro de una clase tienen implícitamente una vinculación externa, puede declararlos con el atributo `dllimport` o `dllexport`, a menos que se exporte toda la clase. Si se importa o se exporta la clase completa, se prohíbe la declaración explícita de funciones miembro y datos como `dllimport` o `dllexport`. Si declara un miembro de datos estático dentro de una definición de clase como `dllexport`, debe producirse una definición en alguna parte dentro del mismo programa (como con la vinculación externa que no es de clase).

De igual forma, puede declarar funciones miembro con los atributos `dllimport` o `dllexport`. En este caso, debe proporcionar una definición de `dllexport` en alguna parte dentro del mismo programa.

Merece la pena comentar varios aspectos importantes con respecto a la importación y exportación selectiva de miembros:

- La importación/exportación selectiva de miembros sirve para proporcionar una versión de la interfaz de clase exportada que es más restrictiva; es decir, una para la que se puede diseñar una DLL que expone menos características públicas y privadas que las que el lenguaje permitiría de otra manera. También es útil para ajustar la interfaz exportable: cuando se sabe que el cliente, por definición, no

puede tener acceso a algunos datos privados, no es necesario exportar toda la clase.

- Si exporta una función virtual de una clase, debe exportarlas todas, o al menos proporcionar versiones que el cliente pueda utilizar directamente.
- Si tiene una clase en la que está utilizando la importación/exportación selectiva de miembros con funciones virtuales, las funciones deben estar en la interfaz exportable o estar definidas insertadas (visibles al cliente).
- Si define un miembro como `__declspec(dllexport)` pero no lo incluye en la definición de clase, se genera un error del compilador. Debe definir el miembro en el encabezado de clase.
- Aunque se permite la definición de miembros de clase como `__declspec(dllimport)` o `__declspec(dllexport)`, no se puede invalidar la interfaz especificada en la definición de clase.
- Si define una función miembro en un lugar distinto del cuerpo de la definición de clase en que se declaró, se genera una advertencia si la función se define como `__declspec(dllexport)` o `__declspec(dllimport)` (si esta definición difiere de la especificada en la declaración de clase).

FIN de Específicos de Microsoft

Consulte también

[__declspec\(dllexport, dllimport\)](#)

empty_bases

Artículo • 16/11/2022 • Tiempo de lectura: 6 minutos

Específicos de Microsoft

El C++ estándar requiere que un objeto más derivado tenga un tamaño distinto de cero y que ocupe uno o varios bytes de almacenamiento. Dado que el requisito solo se extiende a los objetos más derivados, los subobjetos de la clase base no están sujetos a esta restricción. La optimización vacía de clase base (EBCO) aprovecha esta libertad. El resultado es un menor consumo de memoria, lo que puede mejorar el rendimiento. El compilador de Microsoft Visual C++ ha tenido históricamente compatibilidad limitada con EBCO. En la actualización 3 de Visual Studio 2015 y versiones posteriores, hemos agregado un nuevo atributo `_declspec(empty_bases)` para los tipos de clase que aprovecha al máximo esta optimización.

ⓘ Importante

El uso de `_declspec(empty_bases)` puede provocar un cambio importante de ABI en la estructura y el diseño de la clase donde se aplica. Asegúrese de que todo el código del cliente usa las mismas definiciones para estructuras y clases que el código al usar este atributo de clase de almacenamiento.

Sintaxis

```
_declspec( empty_bases )
```

Comentarios

En Visual Studio, en ausencia de alguna especificación `_declspec(align())` o `alignas()`, una clase vacía tiene un tamaño de 1 byte:

C++

```
struct Empty1 {};
static_assert(sizeof(Empty1) == 1, "Empty1 should be 1 byte");
```

Una clase con un único miembro de datos no estático de tipo `char` también tiene un tamaño de 1 byte:

C++

```
struct Struct1
{
    char c;
};

static_assert(sizeof(Struct1) == 1, "Struct1 should be 1 byte");
```

La combinación de estas clases en una jerarquía de clases también da como resultado una clase con un tamaño de 1 byte:

C++

```
struct Derived1 : Empty1
{
    char c;
};

static_assert(sizeof(Derived1) == 1, "Derived1 should be 1 byte");
```

El resultado es la optimización de la clase base vacía en el trabajo, ya que sin él `Derived1` tendría un tamaño de 2 bytes: 1 byte para `Empty1` y 1 byte para `Derived1::c`. El diseño de clase también es óptimo cuando hay una cadena de clases vacías:

C++

```
struct Empty2 : Empty1 {};
struct Derived2 : Empty2
{
    char c;
};

static_assert(sizeof(Derived2) == 1, "Derived2 should be 1 byte");
```

Sin embargo, el diseño de clase predeterminado en Visual Studio no aprovecha las ventajas de EBCO en varios escenarios de herencia:

C++

```
struct Empty3 {};
struct Derived3 : Empty2, Empty3
{
    char c;
};

static_assert(sizeof(Derived3) == 1, "Derived3 should be 1 byte"); // Error
```

Aunque `Derived3` podría tener un tamaño de 1 byte, el diseño de clase predeterminado da como resultado un tamaño de 2 bytes. El algoritmo de diseño de clase agrega 1 byte

de relleno entre dos clases base vacías consecutivas, lo que hace que `Empty2` consuma un byte adicional dentro de `Derived3`:

```
class Derived3 size(2):
+---  
0 | +--- (base class Empty2)
0 | | +--- (base class Empty1)
| | +---  
| +---  
1 | +--- (base class Empty3)
| +---  
1 | c
+---
```

Los efectos de este diseño poco óptimo se componen cuando los requisitos de alineación de una clase base posterior o subobjeto miembro obligan a un relleno adicional:

C++

```
struct Derived4 : Empty2, Empty3
{
    int i;
};

static_assert(sizeof(Derived4) == 4, "Derived4 should be 4 bytes"); // Error
```

La alineación natural de un objeto de tipo `int` es de 4 bytes, por lo que se deben agregar 3 bytes de relleno adicional después de `Empty3` para alinear correctamente a `Derived4::i`:

```
class Derived4 size(8):
+---  
0 | +--- (base class Empty2)
0 | | +--- (base class Empty1)
| | +---  
| +---  
1 | +--- (base class Empty3)
| +---  
| <alignment member> (size=3)
4 | i
+---
```

Otro problema con el diseño de clase predeterminado es que una clase base vacía se puede colocar en un desplazamiento más allá del final de la clase:

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)
| +---
+---
```

Aunque `Struct2` es el tamaño óptimo, `Empty1` se establece en el desplazamiento 1 dentro de `Struct2` pero el tamaño de `Struct2` no aumenta para tenerse en cuenta. Como resultado, para una matriz `A` de objetos `Struct2`, la dirección del subobjeto `Empty1` de `A[0]` será la misma que la dirección de `A[1]`, que no debería ser el caso. Este problema no se produciría si `Empty1` se estableciera en el desplazamiento 0 dentro de `Struct2`, superponiendo así el subobjeto `Struct1`.

El algoritmo de diseño predeterminado no se ha modificado para abordar estas limitaciones y aprovechar completamente a EBCO. Este cambio interrumpiría la compatibilidad binaria. Si el diseño predeterminado de una clase ha cambiado como resultado de EBCO, todos los archivos de objeto y bibliotecas que contengan la definición de clase deberán volver a compilarse para que todos acepten el diseño de clase. Este requisito también se extendería a las bibliotecas obtenidas de orígenes externos. Los desarrolladores de estas bibliotecas tendrían que proporcionar versiones independientes compiladas con y sin el diseño de EBCO para admitir a los clientes que usan diferentes versiones del compilador. Aunque no podemos cambiar el diseño predeterminado, podemos proporcionar un medio para cambiar el diseño por clase con la adición del atributo de clase `__declspec(empty_bases)`. Una clase definida con este atributo puede hacer pleno uso de EBCO.

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  
+---
```

Todos los subobjetos de `Derived3` se colocan en el desplazamiento 0 y su tamaño es el óptimo de 1 byte. Un punto importante que hay que recordar es que `__declspec(empty_bases)` solo afecta al diseño de la clase a la que se aplica. No se aplica recursivamente a las clases base:

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  
| +---  
+---
```

Aunque `__declspec(empty_bases)` se aplica a `Derived5`, no es apto para EBCO porque no tiene clases base vacías directas, por lo que no tiene efecto. Sin embargo, si en su lugar se aplica a la clase base `Derived4`, que es apta para EBCO, tanto `Derived4` como `Derived5` tendrá un diseño óptimo:

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
| +---+
+---
```

Debido al requisito de que todos los archivos de objetos y bibliotecas acepten el diseño de clase, `__declspec(empty_bases)` solo se puede aplicar a las clases que controla. No se puede aplicar a las clases de la biblioteca estándar ni a las clases incluidas en las bibliotecas que tampoco se vuelven a compilar con el diseño EBCO.

FIN de Específicos de Microsoft

Consulte también

[__declspec](#)

Palabras clave

jitintrinsic

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Marca la función como significativa para Common Language Runtime de 64 bits. Se utiliza en algunas funciones de bibliotecas proporcionadas por Microsoft.

Sintaxis

```
__declspec(jitintrinsic)
```

Comentarios

jitintrinsic agrega un objeto MODOPT ([IsJitIntrinsic](#)) a una firma de función.

No se recomienda a los usuarios que utilicen este modificador `__declspec`, ya que pueden producirse resultados inesperados.

Consulte también

[__declspec](#)

[Palabras clave](#)

naked (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Para las funciones declaradas con el atributo `naked`, el compilador genera código sin prólogo ni epílogo. Puede utilizar esta característica para escribir sus propias secuencias de código de prólogo/epílogo mediante código del ensamblador alineado. Las funciones naked son especialmente útiles al escribir controladores de dispositivos virtuales. Tenga en cuenta que el atributo `naked` solo es válido en x86 y ARM, y no está disponible en x64.

Sintaxis

```
__declspec(naked) declarator
```

Comentarios

Dado que el atributo `naked` solo es relevante para la definición de una función y no es un modificador de tipo, las funciones naked deben utilizar la sintaxis de atributos extendida y la palabra clave `__declspec`.

El compilador no puede generar una función inline para una función marcada con el atributo naked, incluso si la función también está marcada con la palabra clave `__forceinline`.

El compilador emite un error si el atributo `naked` se aplica a cualquier cosa que no sea la definición de un método no miembro.

Ejemplos

Este código define una función con el atributo `naked`:

```
__declspec( naked ) int func( formal_parameters ) {}
```

O bien, como alternativa:

```
#define Naked __declspec( naked )
Naked int func( formal_parameters ) {}
```

El atributo `naked` solo afecta a la naturaleza de la generación de código del compilador para las secuencias de prólogo y epílogo de la función. No afecta al código que se genera para llamar a esas funciones. Por tanto, el atributo `naked` no se considera parte del tipo de la función y los punteros a función no pueden tener el atributo `naked`. Además, el atributo `naked` no se puede aplicar a una definición de datos. Por ejemplo, en este ejemplo de código se genera un error:

```
__declspec( naked ) int i;
// Error--naked attribute not permitted on data declarations.
```

El atributo `naked` solo es pertinente para la definición de la función y no se puede especificar en el prototipo de la función. Por ejemplo, esta declaración genera un error del compilador:

```
__declspec( naked ) int func(); // Error--naked attribute not permitted on
function declarations
```

FIN de Específicos de Microsoft

Consulte también

[__declspec](#)

[Palabras clave](#)

[Llamadas de función naked](#)

noalias

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específico de Microsoft

noalias significa que una llamada de función no modifica el estado global visible ni hace referencia a él y solo modifica la memoria designada *directamente* por los parámetros de puntero (direcciónamientos indirectos de primer nivel).

Si una función está anotada como **noalias**, el optimizador puede suponer que, solo los parámetros en sí mismos, dentro de la función y solo los direcciónamientos indirectos de primer nivel de los parámetros del puntero se modifican o se hace referencia a ellos.

La anotación **noalias** solo se aplica dentro del cuerpo de la función anotada. Marcar una función como **_declspec(noalias)** no afecta al alias de punteros devueltos por la función.

Para obtener otra anotación que pueda afectar al alias, consulte [_declspec\(restrict\)](#).

Ejemplo

En el ejemplo siguiente se muestra el uso de **_declspec(noalias)**.

Cuando la función **multiply** que accede a la memoria con está anotada como **_declspec(noalias)**, se le indica al compilador que esta función no modifica el estado global excepto a través de los punteros en su lista de parámetros.

```
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);
}

```

Consulte también

[__declspec](#)

[Palabras clave](#)

`declspec(restrict)`

noinline

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

`__declspec(noinline)` indica al compilador que nunca inserte una función miembro determinada (función en una clase).

Puede merecer la pena no alinear una función si es pequeña y no es crítica para el rendimiento del código. Es decir, si la función es pequeña y no se la llamará a menudo, por ejemplo, una función que controla una condición de error.

Tenga en cuenta que si una función se marca como `noinline`, la función de llamada será más pequeña y, por tanto, será por sí misma un candidato para la alineación del compilador.

C++

```
class X {
    __declspec(noinline) int mbrfunc() {
        return 0;
    } // will not inline
};
```

FIN de Específicos de Microsoft

Consulte también

[__declspec](#)

[Palabras clave](#)

[inline, __inline, __forceinline](#)

noreturn

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Este atributo `__declspec` indica al compilador que una función no devuelve resultados. Como consecuencia, el compilador sabe que no se puede tener acceso al código que sigue a una llamada a una función `__declspec(noreturn)`.

Si el compilador encuentra una función con una ruta de acceso de control que no devuelve un valor, genera una advertencia (C4715) o un mensaje de error (C2202). Si no se puede tener acceso a la ruta de acceso de control debido a una función que nunca devuelve resultados, puede utilizar `__declspec(noreturn)` para evitar esta advertencia o este error.

ⓘ Nota

La adición de `__declspec(noreturn)` a una función que se espera que devuelva resultados puede dar lugar a un comportamiento indefinido.

Ejemplo

En el ejemplo siguiente, la cláusula `else` no contiene una instrucción `return`. La declaración de `fatal` como `__declspec(noreturn)` evita un mensaje de error o de advertencia.

C++

```
// noreturn2.cpp
__declspec(noreturn) extern void fatal () {}

int main() {
    if(1)
        return 1;
    else if(0)
        return 0;
    else
        fatal();
}
```

FIN de Específicos de Microsoft

Consulte también

[_declspec](#)

[Palabras clave](#)

no_sanitize_address

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

El especificador `_declspec(no_sanitize_address)` indica al compilador que deshabilite AddressSanitizer en funciones, variables locales o variables globales. Este especificador se usa junto con [AddressSanitizer](#).

ⓘ Nota

`_declspec(no_sanitize_address)` deshabilita el comportamiento del *compilador*, no el comportamiento del *runtime*.

Ejemplo

Vea la [referencia de compilación de AddressSanitizer](#) para obtener ejemplos.

FIN de Específicos de Microsoft

Consulte también

[_declspec](#)

[Palabras clave](#)

[AddressSanitizer](#)

nothrow (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Atributo extendido `__declspec` que se puede usar en la declaración de funciones.

Sintaxis

```
return-type __declspec(nothrow) [call-convention] function-name ([argument-list])
```

Comentarios

Se recomienda que todo el código nuevo use el `noexcept` operador en lugar de `__declspec(nothrow)`.

Este atributo indica al compilador que la función declarada y las funciones a las que llama nunca iniciarán una excepción. Sin embargo, no aplica la directiva. En otras palabras, nunca hace que `std::terminate` se invoque, a diferencia de `noexcept`, o en modo `std:c++17` (Visual Studio 2017 versión 15.5 y posteriores), `throw()`.

Con el modelo de control asincrónico de excepciones, que ahora es el predeterminado, el compilador puede eliminar los mecanismos de seguimiento de la duración de algunos objetos que no se pueden desenredar en esa función, y reducir significativamente el tamaño del código. Dada la directiva de preprocesador siguiente, las tres declaraciones de función que se muestran a continuación son equivalentes en modo `/std:c++14`:

C++

```
#define WINAPI __declspec(nothrow) __stdcall

void WINAPI f1();
void __declspec(nothrow) __stdcall f2();
void __stdcall f3() throw();
```

En el modo `/std:c++17`, `throw()` no es equivalente a los demás que usan `__declspec(nothrow)` porque hace que `std::terminate` se invoque si se produce una excepción desde la función.

La `void __stdcall f3() throw();` declaración, usa la sintaxis definida en el estándar de C++. En C++17, la palabra clave `throw()` está en desuso.

Consulte también

[_declspec](#)

[noexcept](#)

[Palabras clave](#)

novtable

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Este es un atributo extendido `__declspec`.

Esta forma de `__declspec` se puede aplicar a cualquier declaración de clase, pero solo se debe aplicar a clases de interfaz puras, es decir, clases que nunca crearán instancias por sí solas. El objeto `__declspec` impide que el compilador genere código para inicializar vptr en los constructores y el destructor de la clase. En muchos casos, esto quita las únicas referencias a la vtable asociadas a la clase y, en consecuencia, el vinculador la quita. El uso de esta forma de `__declspec` puede producir una reducción significativa del tamaño del código.

Si intenta crear una instancia de una clase marcada con `novtable` y, a continuación, tener acceso a un miembro de clase, recibirá una infracción de acceso (AV).

Ejemplo

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

FIN de Específicos de Microsoft

Consulte también

[_declspec](#)

[Palabras clave](#)

process

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Especifica que el proceso de la aplicación administrada debe tener una única copia de una variable global determinada, una variable miembro estática o una variable local estática compartida en todos los dominios de aplicación del proceso. Esto estaba pensado principalmente para usarse al compilar con `/clr:pure`, que está en desuso en Visual Studio 2015 y no se admite en Visual Studio 2017. Cuando se compila con `/clr`, las variables globales y estáticas son por proceso de forma predeterminada (no tiene que usar `__declspec(process)`).

Solo una variable global, una variable miembro estática o una variable local estática de tipo nativo se pueden marcar con `__declspec(process)`.

`process` solo es válido al compilar con `/clr`.

Si desea que cada dominio de aplicación tenga su propia copia de una variable global, utilice [appdomain](#).

Consulte [Dominios de aplicación y Visual C++](#) para obtener más información.

Consulte también

[__declspec](#)

[Palabras clave](#)

propiedad (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Este atributo se puede aplicar a los "miembros de datos virtuales" no estáticos en una definición de clase o estructura. El compilador trata a estos "miembros de datos virtuales" como miembros de datos y cambia sus referencias en las llamadas de función.

Sintaxis

```
__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
```

Comentarios

Cuando el compilador consulta un miembro de datos declarado con este atributo en el lado derecho de un operador de selección de miembros ("." o "->"), convierte la operación a una función `get` o `put`, en función de que la expresión sea un valor L o un valor R. En contextos más complicados, como "`+=`", se realiza una reescritura y se usan ambos, `get` y `put`.

Este atributo se puede utilizar también en la declaración de una matriz vacía en una definición de clase o estructura. Por ejemplo:

C++

```
__declspec(property(get=GetX, put=PutX)) int x[];
```

La instrucción anterior indica que `x[]` se puede utilizar con uno o más índices de matriz. En este caso, `i=p->x[a][b]` se convertirá en `i=p->GetX(a, b)` y `p->x[a][b] = i` se convertirá en `p->PutX(a, b, i);`

FIN de Específicos de Microsoft

Ejemplo

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;
}
```

Consulte también

[__declspec](#)

[Palabras clave](#)

restrict

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Cuando se aplica a una declaración de función o a una definición que devuelve un tipo de puntero, `restrict` indica al compilador que la función devuelve un objeto que no tiene *alias*, es decir, que no hace referencia a él ningún otro puntero. Esto permite al compilador realizar optimizaciones adicionales.

Sintaxis

```
| _declspec(restrict) pointer_return_type function();
```

Comentarios

El compilador propaga `_declspec(restrict)`. Por ejemplo, la función `malloc` de CRT tiene una decoración `_declspec(restrict)` y, por lo tanto, el compilador supone que los punteros que `malloc` inicializa en ubicaciones de memoria tampoco tienen alias de punteros existentes anteriormente.

El compilador no comprueba que el puntero devuelto realmente no tenga alias. Es responsabilidad del programador garantizar que el programa no use un alias de un puntero marcado con el modificador `restrict _declspec`.

Para conocer la semántica similar en las variables, consulte [_restrict](#).

Para obtener otra anotación que se aplica al alias dentro de una función, consulte [_declspec\(noalias\)](#).

Para obtener información sobre la palabra clave `restrict` que forma parte de C++ AMP, consulte [restrict \(C++ AMP\)](#).

Ejemplo

En el ejemplo siguiente se muestra el uso de `_declspec(restrict)`.

Cuando `_declspec(restrict)` se aplica a una función que devuelve un puntero, esto indica al compilador que la memoria a la que apunta el valor devuelto no tiene alias. En este ejemplo, los punteros `mempool` y `memptr` son globales, por lo que el compilador no

puede estar seguro de que la memoria a la que hacen referencia no tiene alias. Aun así, se usan en `ma` y su llamador `init` de una manera que devuelve la memoria a la que el programa no hace referencia de otro modo, por lo que `_declspec(restrict)` se usa para ayudar al optimizador. Esto es similar a la forma en que los encabezados de CRT decoran funciones de asignación como `malloc` mediante `_declspec(restrict)` para indicar que siempre devuelven memoria que no puede tener alias de punteros existentes.

```
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);  
}
```

FIN de Específicos de Microsoft

Consulte también

Palabras clave

[_declspec](#)
[_declspec\(noalias\)](#)

safebuffers

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Indica al compilador que no inserte comprobaciones de seguridad de saturación del búfer para una función.

Sintaxis

```
__declspec( safebuffers )
```

Comentarios

La opción del compilador **/GS** hace que el compilador compruebe las saturaciones del búfer mediante la inserción de comprobaciones de seguridad en la pila. Los tipos de estructuras de datos que son aptos para las comprobaciones de seguridad se describen en [/GS \(Comprobación de seguridad del búfer\)](#). Para obtener más información sobre la detección de saturación del búfer, consulte [Características de seguridad en MSVC](#).

Una revisión manual externo o realizado por un experto para revisar el código puede determinar si una función está protegida de la saturación del búfer. En ese caso, se pueden suprimir las comprobaciones de seguridad de una función mediante la aplicación de la palabra clave **__declspec(safebuffers)** en la declaración de función.

⊗ Precaución

Las comprobaciones de seguridad del búfer proporcionan una protección de seguridad importante y apenas repercuten en el rendimiento. Por tanto, se recomienda que no las suprima, excepto en el caso poco frecuente de que el rendimiento de una función tenga una importancia crítica y se sepa que la función está segura.

Funciones insertadas

Una *función principal* puede utilizar una palabra clave [inlining](#) para insertar una copia de una *función secundaria*. Si la palabra clave `_declspec(safebuffers)` se aplica a una función, la detección de saturación del búfer se suprime para esa función. Sin embargo, inlining afecta a la palabra clave `_declspec(safebuffers)` de las maneras siguientes.

Supongamos que la opción del compilador `/GS` está especificada para ambas funciones pero la función principal especifica la palabra clave `_declspec(safebuffers)`. Las estructuras de datos en la función secundaria hacen que sea apta para las comprobaciones de seguridad, y la función no suprime dichas comprobaciones. En este caso:

- Especifique la palabra clave `_forceinline` en la función secundaria para hacer que el compilador inserte esa función independientemente de las optimizaciones del compilador.
- Dado que la función secundaria es apta para las comprobaciones de seguridad, estas últimas también se aplican a la función principal, aunque especifique la palabra clave `_declspec(safebuffers)`.

Ejemplo

En el código siguiente se muestra cómo se puede usar la palabra clave `_declspec(safebuffers)`.

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;
}
```

Consulte también

[_declspec](#)

[Palabras clave](#)

[inline, __inline, __forceinline](#)

[strict_gs_check](#)

selectany

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Indica al compilador que el elemento de datos globales declarado (variable u objeto) es un COMDAT de selección (una función empaquetada).

Sintaxis

```
| __declspec( selectany ) declarador
```

Comentarios

En tiempo de vinculación, si se ven varias definiciones de un COMDAT, el vinculador selecciona una y descarta el resto. Si se selecciona la opción del enlazador [/OPT:REF](#) (Optimizaciones), se producirá la eliminación de COMDAT para quitar todos los elementos de datos no referenciados de la salida del enlazador.

Los constructores y la asignación mediante una función global o métodos estáticos en la declaración no crean una referencia y no impedirán la eliminación de /OPT:REF. No se debe depender de los efectos secundarios de ese código si no existen otras referencias a los datos.

Para los objetos globales inicializados dinámicamente, **selectany** también descartará un código de inicialización de objeto no referenciado.

Un elemento de datos globales se puede inicializar normalmente solo una vez en un proyecto EXE o DLL. **selectany** se puede usar en la inicialización de datos globales definidos por los encabezados cuando el mismo encabezado aparece en más de un archivo de código fuente. **selectany** está disponible en los compiladores de C y C++.

ⓘ Nota

selectany solo se puede aplicar a la inicialización real de elementos de datos globales externamente visibles.

Ejemplo: atributo **selectany**

En este código se muestra cómo usar el atributo `selectany`:

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);
```

Ejemplo: uso del atributo `selectany` para garantizar el plegamiento de COMDAT de datos

En este código se muestra cómo usar el atributo `selectany` para garantizar el plegamiento de COMDAT de datos cuando también se usa la opción del enlazador `/OPT:ICF`. Observe que los datos deben marcarse con `selectany` y colocarse en una sección `const` (de solo lectura). Debe especificar explícitamente la sección de solo lectura.

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;
}
```

FIN de Específicos de Microsoft

Consulte también

[__declspec](#)

[Palabras clave](#)

spectre

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Indica al compilador que no inserte instrucciones de barrera de ejecución especulativa de la variante 1 de Spectre para una función.

Sintaxis

```
| __declspec( spectre(nomitigation) )
```

Comentarios

La opción del compilador [/Qspectre](#) hace que el compilador inserte instrucciones de barrera de ejecución especulativa. Se insertan donde el análisis indica que existe una vulnerabilidad de seguridad de la variante 1 de Spectre. Las instrucciones específicas emitidas dependen del procesador. Aunque estas instrucciones deben tener un impacto mínimo en el tamaño o el rendimiento del código, puede haber casos en los que el código no se vea afectado por la vulnerabilidad y requiera un rendimiento máximo.

El análisis experto puede determinar que una función es segura de un defecto de derivación de comprobación de límites de la variante 1 de Spectre. En ese caso, se puede suprimir la generación de código de mitigación dentro de una función aplicando `__declspec(spectre(nomitigation))` a la declaración de función.

⊗ Precaución

Las instrucciones de la barrera de ejecución especulativa [/Qspectre](#) proporcionan una protección de seguridad importante y tienen un efecto insignificante en el rendimiento. Por tanto, se recomienda que no las suprima, excepto en el caso poco frecuente de que el rendimiento de una función tenga una importancia crítica y se sepa que la función está segura.

Ejemplo

En el siguiente código se muestra cómo usar `__declspec(spectre(nomitigation))`:

C++

```
// compile with: /c /Qspectre
static __declspec(spectre(nomitigation))
int noSpectreIssues() {
    // No Spectre variant 1 vulnerability here
    // ...
    return 0;
}

int main() {
    noSpectreIssues();
    return 0;
}
```

FIN de Específicos de Microsoft

Consulte también

[__declspec](#)

[Palabras clave](#)

[/Qspectre](#)

subproceso

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

Específicos de Microsoft

El modificador de clase de almacenamiento extendida `thread` se usa para declarar una variable local para el subprocesso. Para el portable equivalente en C++11 y versiones posteriores, use el especificador de clase de almacenamiento para el código portable `thread_local`. En Windows `thread_local` se implementa con `__declspec(thread)`.

Sintaxis

`__declspec(thread) declarador`

Comentarios

El almacenamiento local para el subprocesso (TLS) es el mecanismo por el que cada subprocesso de un proceso con varios subprocessos asigna almacenamiento para los datos específicos de ese subprocesso. En los programas multiproceso estándar, los datos se comparten entre todos los subprocessos de un proceso dado, mientras que el almacenamiento local para el subprocesso es el mecanismo para asignar datos por subprocesso. Para obtener un análisis completo de los subprocessos, consulte [Multithreading](#).

Las declaraciones de variables locales para el subprocesso deben utilizar la [sintaxis de atributos extendida](#) y la palabra clave `__declspec` con la palabra clave `thread`. Por ejemplo, el código siguiente declara una variable local de subprocesso de entero y la inicializa con un valor:

C++

```
__declspec( thread ) int tls_i = 1;
```

Al usar variables locales de subprocesso en bibliotecas cargadas dinámicamente, se debe tener en cuenta los factores que pueden hacer que una variable local de subprocesso no se inicialice correctamente:

1. Si la variable se inicializa con una llamada de función (incluidos los constructores), solo se llamará a esta función para el subprocesso que causó la carga del archivo binario o DLL en el proceso y para esos subprocessos que se iniciaron después de

cargar el archivo binario o DLL. No se llama a las funciones de inicialización para ningún otro subprocesso que ya se estaba ejecutando cuando se cargó el archivo DLL. La inicialización dinámica se produce en la llamada DllMain para DLL_THREAD_ATTACH, pero el archivo DLL nunca obtiene ese mensaje si el archivo DLL no está en el proceso cuando se inicia el subprocesso.

2. Las variables locales de subprocesso que se inicializan estáticamente con valores constantes se inicializan normalmente correctamente en todos los subprocessos. Sin embargo, a partir de diciembre de 2017 hay una incidencia de conformidad conocida en el compilador de Microsoft C++ en el que las variables `constexpr` reciben la inicialización dinámica en lugar de estática.

Nota: Se espera que ambos problemas se corrijan en futuras actualizaciones del compilador.

Además, se deben seguir estas instrucciones cuando se declaren objetos y variables locales para el subprocesso:

- Solo se puede aplicar el atributo `thread` a las declaraciones y definiciones de datos y a clases; `thread` no se puede usar en declaraciones o definiciones de función.
- Solo se puede especificar el atributo `thread` en elementos de datos con duración de almacenamiento estática. Entre ellos se incluyen objetos de datos globales (tanto `static` como `extern`), objetos estáticos locales y miembros de datos estáticos de clases. No se pueden declarar objetos de datos automáticos con el atributo `thread`.
- Se debe usar el atributo `thread` para la declaración y definición de un objeto local para el subprocesso, tanto si la declaración y la definición se realizan en el mismo archivo como en archivos distintos.
- No se puede usar el atributo `thread` como modificador de tipo.
- Como la declaración de objetos que usan el atributo `thread` está permitida, los dos ejemplos siguientes son semánticamente equivalentes:

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.
```

- El estándar de C permite la inicialización de un objeto o de una variable con una expresión que contenga una referencia a sí misma, pero solo para objetos no estáticos. Aunque C++ normalmente permite esa inicialización dinámica de objetos con una expresión que contenga una referencia a sí misma, este tipo de inicialización no se permite con objetos locales de un subproceso. Por ejemplo:

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++
```

Una expresión `sizeof` que incluye el objeto que se está inicializando no representa una referencia a sí misma y se permite en C y en C++.

FIN de Específicos de Microsoft

Consulte también

[_declspec](#)

[Palabras clave](#)

[Almacenamiento local de subprocesos \(TLS\)](#)

uuid (C++)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

El compilador adjunta un GUID a una clase o estructura declarada o definida (solo definiciones completas de objeto COM) con el atributo `uuid`.

Sintaxis

```
__declspec( uuid("ComObjectGUID") ) declarator
```

Comentarios

El atributo `uuid` toma una cadena como argumento. Esta cadena denomina un GUID en formato del Registro normal con o sin los delimitadores { }. Por ejemplo:

C++

```
struct __declspec(uuid("00000000-0000-0000-c000-000000000046")) IUnknown;
struct __declspec(uuid("{00020400-0000-0000-c000-000000000046}") IDispatch;
```

Este atributo se puede aplicar en una nueva declaración. Esto permite que los encabezados de sistema suministren las definiciones de interfaces tales como `IUnknown`, y que la nueva declaración en algún otro encabezado (por ejemplo, <comdef.h>) suministre el GUID.

La palabra clave `_uuidof` se puede aplicar para recuperar el GUID de constante asociado a un tipo definido por el usuario.

FIN de Específicos de Microsoft

Consulte también

[_declspec](#)

[Palabras clave](#)

restrict

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Al igual que el modificador `_declspec` (`restrict`), la `_restrict` palabra clave (dos caracteres de subrayado iniciales "_") indica que un símbolo no está con alias en el ámbito actual. La palabra clave `_restrict` difiere del modificador `_declspec` (`restrict`) en lo siguiente:

- La palabra clave `_restrict` solo es válida en variables y solo `_declspec` (`restrict`) es válido en declaraciones de función y definiciones.
- `_restrict` es similar a `restrict` para C a partir de C99 y disponible en [/std:c11 modo](#) o [/std:c17](#), pero `_restrict` se puede usar en programas de C++ y C.
- Cuando se utiliza `_restrict`, el compilador no propagará la propiedad sin alias de una variable. Es decir, si se asigna una variable `_restrict` a una variable que no es `_restrict`, el compilador seguirá permitiendo que la variable que no es `_restrict` tenga un alias. Este comportamiento es diferente del de la palabra clave `restrict` del lenguaje C C99.

Normalmente, si se modifica el comportamiento de una función completa, es mejor usar `_declspec` (`restrict`) que la palabra clave.

A efectos de compatibilidad con versiones anteriores, `_restrict` es un sinónimo de `_restrict` a menos que se especifique la opción del compilador [/Za \(Deshabilitar extensiones de lenguaje\)](#).

En Visual Studio 2015 y versiones posteriores, se puede usar `_restrict` en las referencias de C++.

ⓘ Nota

Cuando se utiliza en una variable que tiene también la palabra clave `volatile`, `volatile` tendrá prioridad.

Ejemplo

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;
};
```

Consulte también

[Palabras clave](#)

__sptr, __uptr

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

Específicos de Microsoft

Use el modificador **__sptr** o **__uptr** en una declaración de puntero de 32 bits para especificar cómo convierte el compilador un puntero de 32 bits en un puntero de 64 bits. Un puntero de 32 bits se convierte, por ejemplo, cuando se asigna a una variable de puntero de 64 bits o se deshace la referencia en una plataforma de 64 bits.

La documentación de Microsoft para la compatibilidad con plataformas de 64 bits hace referencia a veces al bit más significativo de un puntero de 32 bits como el bit de signo. De forma predeterminada, el compilador utiliza la extensión de signo para convertir un puntero de 32 bits en un puntero de 64 bits. Es decir, los 32 bits menos significativos del puntero de 64 bits se establecen en el valor del puntero de 32 bits y los 32 bits más significativos se establecen en el valor del bit de signo del puntero de 32 bits. Esta conversión produce resultados correctos si el bit de signo es 0, pero no si el bit de signo es 1. Por ejemplo, la dirección de 32 bits 0x7FFFFFFF produce la dirección de 64 bits equivalente 0x00000007FFFFFFF, pero la dirección de 32 bits 0x80000000 cambia incorrectamente a 0xFFFFFFFF80000000.

El modificador **__sptr** o de puntero con signo especifica que una conversión de puntero establezca los bits más significativos de un puntero de 64 bits en el bit de signo del puntero de 32 bits. El modificador **__uptr** o de puntero sin signo especifica que una conversión establezca los bits más significativos en cero. Las declaraciones siguientes muestran los modificadores **__sptr** y **__uptr** usados con dos punteros incompletos, dos punteros completos con el tipo **__ptr32** y un parámetro de función.

C++

```
int * __sptr psp;
int * __uptr pup;
int * __ptr32 __sptr psp32;
int * __ptr32 __uptr pup32;
void MyFunction(char * __uptr __ptr32 myValue);
```

Use los modificadores **__sptr** y **__uptr** con declaraciones de puntero. Use los modificadores en la posición de un [calificador de tipo de puntero](#), lo que significa que el modificador debe seguir el asterisco. No se puede usar los modificadores con [punteros a miembros](#). Los modificadores no afectan a las declaraciones que no son de puntero.

Por compatibilidad con versiones anteriores, `_sptr` y `_uptr` son sinónimos de `__sptr` y `__uptr` a menos que se especifique la opción del compilador [/Za \(Deshabilitar extensiones de lenguaje\)](#).

Ejemplo

El ejemplo siguiente declara punteros de 32 bits que usan los modificadores `__sptr` y `__uptr`, además asigna cada puntero de 32 bits a una variable de puntero de 64 bits y muestra el valor hexadecimal de cada puntero de 64 bits. El ejemplo se compila con el compilador de 64 bits nativo y se ejecuta en una plataforma de 64 bits.

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
```

FIN de Específicos de Microsoft

Consulte también

[Modificadores específicos de Microsoft](#)

`_unaligned`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específico de Microsoft. Cuando se declara un puntero con el modificador `_unaligned`, el compilador supone que el puntero hace referencia a datos no alineados. Por lo tanto, se genera código adecuado para la plataforma para controlar las lecturas y escrituras no alineadas a través del puntero.

Comentarios

Este modificador describe solamente la alineación de los datos objetivo; se supone que el puntero en sí está alineado.

La necesidad de la palabra clave `_unaligned` varía según la plataforma y el entorno. Si los datos no se marcan correctamente, se pueden producir problemas que van desde mermas en el rendimiento hasta errores de hardware. El modificador `_unaligned` no se puede usar en la plataforma x86.

A efectos de compatibilidad con versiones anteriores, `_unaligned` es un sinónimo de `_unaligned` a menos que se especifique la opción del compilador [/Za \(Deshabilitar extensiones de lenguaje\)](#).

Para obtener más información sobre la alineación, vea:

- [align](#)
- [Operador alignof](#)
- [pack](#)
- [/Zp \(Alineación de miembros de estructura\)](#)
- [Ejemplos de alineación de estructuras x64](#)

Consulte también

[Palabras clave](#)

w64

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Esta palabra clave específica de Microsoft está obsoleta. En versiones de Visual Studio anteriores a Visual Studio 2013, le permite marcar variables, de modo que cuando usted compile con [/Wp64](#) el compilador notifique cualquier advertencia que se notificaría si estuviese compilando con un compilador de 64 bits.

Sintaxis

`type __w64 identifier`

Parámetros

type

Uno de los tres tipos que pueden causar problemas al migrar el código de un compilador de 32 bits a otro de 64 bits: `int`, `long` o un puntero.

identifier

Identificador de la variable que va a crear.

Comentarios

Importante

La opción del compilador [/Wp64](#) y la palabra clave `__w64` están en desuso en Visual Studio 2010 y Visual Studio 2013 y se eliminaron a partir de Visual Studio 2013. Si usa la opción del compilador `/Wp64` en la línea de comandos, el compilador emite la advertencia D9002 de la línea de comandos. La palabra clave `__w64` se ignora de modo silencioso. En lugar de usar esta opción y la palabra clave para detectar problemas de portabilidad a 64 bits, use un compilador de Microsoft C++ cuyo destino sea una plataforma de 64 bits. Para obtener más información, consulte [Configuración de Visual C++ para destinos x64 de 64 bits](#).

Cualquier definición de tipo que contenga `__w64` debe ser de 32 bits en x86 y de 64 bits en x64.

Para detectar problemas de portabilidad mediante versiones del compilador de Microsoft C++ anteriores a Visual Studio 2010, se debe especificar la palabra clave `__w64` en cualquier definición de tipo que cambie de tamaño entre plataformas de 32 y 64 bits. Para dicho tipo, `__w64` únicamente debe aparecer en la definición de 32 bits de la definición de tipo.

Por compatibilidad con versiones anteriores, `_w64` es sinónimo de `__w64`, a menos que se especifique la opción del compilador [/Za \(deshabilitar extensiones de lenguaje\)](#).

Se omite la palabra clave `__w64` si la compilación no usa `/Wp64`.

Para obtener más información sobre la portabilidad a 64 bits, vea los temas siguientes:

- [Opciones del compilador de MSVC](#)
- [Migración de código de 32 bits a código de 64 bits](#)
- [Configuración de Visual C++ en destinos de 64 bits, x64](#)

Ejemplo

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
}
```

Consulte también

[Palabras clave](#)

__func__

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

(C++11) La `__func__` del identificador predefinido se define implícitamente como una cadena que contiene el nombre no completo y sin adornar de la función de inclusión. El estándar de C++ exige el uso de `__func__`, que no es una extensión de Microsoft.

Sintaxis

C++

```
__func__
```

Valor devuelto

Devuelve una matriz const char terminada en null de caracteres que contiene el nombre de función.

Ejemplo

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;
}
```

Requisitos

C++11

Compatibilidad con COM del compilador

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

El compilador de Microsoft C++ puede leer directamente bibliotecas de tipos del Modelo de objetos componentes (COM) y traducir el contenido a código fuente de C++ que se puede incluir en la compilación. Existen extensiones de lenguaje disponibles para facilitar la programación COM en el cliente para aplicaciones de escritorio.

Mediante el uso de la [directiva de preprocesador #import](#), el compilador puede leer una biblioteca de tipos y convertirla en un archivo de encabezado de C++ que describe las interfaces COM como clases. Existe un conjunto de atributos `#import` disponible para el control por parte del usuario del contenido de los archivos de encabezado de biblioteca de tipos resultantes.

Puede usar el `uuid` de atributo extendido `_declspec` para asignar un identificador único global (GUID) a un objeto COM. Se puede usar la palabra clave `_uuidof` para extraer el GUID asociado a un objeto COM. Se puede usar otro atributo `_declspec`, [property](#), para especificar los métodos `get` y `set` para un miembro de datos de un objeto COM.

Se proporciona un conjunto de clases y funciones globales de compatibilidad con COM para admitir los tipos `VARIANT` y `BSTR`, implementar punteros inteligentes y encapsular el objeto de error iniciado por `_com_raise_error`:

- [Funciones globales COM del compilador](#)
- [_bstr_t](#)
- [_com_error](#)
- [_com_ptr_t](#)
- [_variant_t](#)

FIN de Específicos de Microsoft

Consulte también

[Clases de compatibilidad con COM del compilador](#)
[Funciones globales COM del compilador](#)

Funciones globales COM de compilador

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Están disponibles las siguientes rutinas:

| Función | Descripción |
|--|---|
| _com_raise_error | Produce un _com_error en respuesta a un error. |
| _set_com_error_handler | Reemplaza la función predeterminada que se utiliza para el control de errores de COM. |
| ConvertBSTRToString | Convierte un valor <code>BSTR</code> en un objeto <code>char *</code> . |
| ConvertStringToBSTR | Convierte un valor <code>char *</code> en un objeto <code>BSTR</code> . |

FIN de Específicos de Microsoft

Consulte también

[Clases de compatibilidad con COM del compilador](#)

[Compatibilidad con COM del compilador](#)

_com_raise_error

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Produce un [_com_error](#) en respuesta a un error.

Sintaxis

C++

```
void __stdcall _com_raise_error(
    HRESULT hr,
    IErrorInfo* perrinfo = 0
);
```

Parámetros

h

Información de HRESULT.

perrinfo

Objeto [IErrorInfo](#).

Comentarios

`_com_raise_error`, que se define en `<comdef.h>`, se puede reemplazar por una versión escrita por el usuario con el mismo nombre y prototipo. Esto se podría hacer si se desea utilizar `#import` pero no se desea utilizar el control de excepciones de C++. En ese caso, una versión de usuario de `_com_raise_error` podría decidir hacer un `longjmp` o mostrar un cuadro de mensaje y detenerse. La versión de usuario no debe volver, sin embargo, porque el código de compatibilidad con COM del compilador no espera que vuelva.

También puedes utilizar [_set_com_error_handler](#) para reemplazar la función predeterminada de control de errores.

De forma predeterminada, `_com_raise_error` se define de la siguiente manera:

C++

```
void __stdcall _com_raise_error(HRESULT hr, IErrorInfo* perrinfo) {
    throw _com_error(hr, perrinfo);
}
```

FIN de Específicos de Microsoft

Requisitos

Encabezado:<comdef.h>

Lib: si la opción del compilador **wchar_t is Native Type** está activada, use comsuppw.lib o comsuppwd.lib. Si **wchar_t is Native Type** está desactivada, use comsupp.lib. Para obtener más información, vea [/Zc:wchar_t \(wchar_t es un tipo nativo\)](#).

Consulte también

[Funciones globales COM del compilador](#)
[_set_com_error_handler](#)

ConvertStringToBSTR

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Convierte un valor `char *` en un objeto `BSTR`.

Sintaxis

```
BSTR __stdcall ConvertStringToBSTR(const char* pSrc)
```

Parámetros

pSrc

Una variable `char *`.

Ejemplo

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
```

FIN de Específicos de Microsoft

Requisitos

Encabezado:<comutil.h>

Biblioteca: comsuppw.lib o comsuppwd.lib (vea [/Zc:wchar_t \(wchar_t es tipo nativo\)](#) para obtener más información)

Consulte también

[Funciones globales COM del compilador](#)

ConvertBSTRToString

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Convierte un valor `BSTR` en un objeto `char *`.

Sintaxis

```
char* __stdcall ConvertBSTRToString(BSTR pSrc);
```

Parámetros

pSrc

Una variable `BSTR`.

Comentarios

`ConvertBSTRToString` asigna una cadena que debe eliminar.

Ejemplo

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
```

FIN de Específicos de Microsoft

Requisitos

Encabezado:<comutil.h>

Biblioteca: comsuppw.lib o comsuppwd.lib (vea [/Zc:wchar_t \(wchar_t es tipo nativo\)](#) para obtener más información)

Consulte también

[Funciones globales COM del compilador](#)

_set_com_error_handler

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Reemplaza la función predeterminada que se utiliza para el control de errores de COM. `_set_com_error_handler` es Microsoft-specific.

Sintaxis

C++

```
void __stdcall _set_com_error_handler(
    void (__stdcall *pHandler)(
        HRESULT hr,
        IErrorInfo* perrinfo
    )
);
```

Parámetros

pHandler

Puntero a la función de reemplazo.

h

Información de HRESULT.

perrinfo

Objeto `IErrorInfo`.

Comentarios

De manera predeterminada, `_com_raise_error` controla todos los errores de COM. Puede cambiar este comportamiento mediante `_set_com_error_handler` para llamar a su propia función de control de errores.

La función de reemplazo debe tener una firma que sea equivalente a la de `_com_raise_error`.

Ejemplo

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!

Requisitos

Header:<comdef.h>

Lib: si se especifica la opción del compilador **/Zc:wchar_t** (valor predeterminado), use comsuppw.lib o comsuppwd.lib. Si se especifica la opción del compilador **/Zc:wchar_t-**, use comsupp.lib. Para obtener más información, incluso cómo establecer esta opción en el IDE, vea [/Zc:wchar_t \(wchar_t es un tipo nativo\)](#).

Consulte también

Funciones globales COM del compilador

Clases de compatibilidad con COM del compilador

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Las clases estándar se utilizan para proporcionar compatibilidad con algunos de los tipos COM. Las clases se definen en los archivos `<comdef.h>` y de encabezado generados desde la biblioteca de tipos.

| Clase | Propósito |
|-------------------------|--|
| <code>_bstr_t</code> | Encapsula el tipo <code>BSTR</code> para proporcionar operadores y métodos útiles. |
| <code>_com_error</code> | Define el objeto de error producido por <code>_com_raise_error</code> en la mayoría de los errores. |
| <code>_com_ptr_t</code> | Encapsula punteros de interfaz COM y automatiza las llamadas necesarias a <code>AddRef</code> , <code>Release</code> y <code>QueryInterface</code> . |
| <code>_variant_t</code> | Encapsula el tipo <code>VARIANT</code> para proporcionar operadores y métodos útiles. |

FIN de Específicos de Microsoft

Consulte también

[Compatibilidad con COM del compilador](#)

[Funciones globales COM del compilador](#)

[Referencia del lenguaje C++](#)

Clase `_bstr_t`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Un objeto `_bstr_t` encapsula el [tipo de datos BSTR](#). La clase administra la asignación y desasignación de recursos con llamadas de función a `SysAllocString` y `SysFreeString` y otras API de `BSTR` cuando es necesario. La clase `_bstr_t` utiliza el recuento de referencias para evitar una sobrecarga excesiva.

Miembros

Construcción

| Constructor | Descripción |
|----------------------|--|
| <code>_bstr_t</code> | Construye un objeto <code>_bstr_t</code> . |

Operaciones

| Función | Descripción |
|-------------------------|---|
| <code>Assign</code> | Copia un valor <code>BSTR</code> en el valor <code>BSTR</code> contenido en <code>_bstr_t</code> . |
| <code>Attach</code> | Vincula un contenedor <code>_bstr_t</code> a un <code>BSTR</code> . |
| <code>copy</code> | Crea una copia del objeto <code>BSTR</code> encapsulado. |
| <code>Detach</code> | Devuelve el <code>BSTR</code> contenido en <code>_bstr_t</code> y desasocia <code>BSTR</code> de <code>_bstr_t</code> . |
| <code>GetAddress</code> | Apunta al <code>BSTR</code> contenido en <code>_bstr_t</code> . |
| <code>GetBSTR</code> | Señala al principio del objeto <code>BSTR</code> incluido en <code>_bstr_t</code> . |
| <code>length</code> | Devuelve el número de caracteres de <code>_bstr_t</code> . |

Operadores

| Operador | Descripción |
|-------------------------|---|
| <code>operator =</code> | Asigna un nuevo valor a un objeto <code>_bstr_t</code> existente. |

| Operador | Descripción |
|--|--|
| <code>operator +=</code> | Agrega caracteres al final del objeto <code>_bstr_t</code> . |
| <code>operator +</code> | Concatena dos cadenas. |
| <code>operator !</code> | Comprueba si el <code>BSTR</code> encapsulado es una cadena NULL. |
| <code>operator ==</code> <code>operator !=</code> <code>operator <</code> <code>operator ></code> <code>operator <=</code> <code>operator >=</code> | Compara dos objetos <code>_bstr_t</code> . |
| <code>operator wchar_t*</code> <code>operator char*</code> | Extrae los punteros al objeto <code>BSTR</code> multibyte o Unicode encapsulado. |

FIN de Específicos de Microsoft

Requisitos

Encabezado:<comutil.h>

Lib:`comsuppw.Lib` o `comsuppwd.Lib` (Para obtener más información, consulte [/Zc:wchar_t \(wchar_t es tipo nativo\)](#))

Consulte también

[Clases de compatibilidad con COM del compilador](#)

`_bstr_t::_bstr_t`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Construye un objeto `_bstr_t`.

Sintaxis

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
);
```

Parámetros

`s1`

Objeto `_bstr_t` que se va a copiar.

`s2`

Cadena multibyte.

`s3`

Cadena Unicode.

`var`

Objeto `_variant_t`.

`bstr`

Objeto `BSTR` existente.

`fCopy`

Si es `false`, el argumento `bstr` se adjunta al nuevo objeto sin crear una copia mediante una llamada a `SysAllocString`.

Comentarios

La clase `_bstr_t` proporciona varios constructores:

`_bstr_t()`

Construye un objeto `_bstr_t` predeterminado que encapsula un objeto `BSTR` null.

`_bstr_t(_bstr_t& s1)`

Construye un objeto `_bstr_t` como copia de otro. Este constructor realiza una copia *superficial*, que incrementa el recuento de referencias del objeto `BSTR` encapsulado, en lugar de crear uno nuevo.

`_bstr_t(char* s2)`

Construye un objeto `_bstr_t` mediante una llamada a `SysAllocString` para crear un nuevo objeto `BSTR` y, a continuación, lo encapsula. Este constructor realiza primero una conversión de multibyte a Unicode.

`_bstr_t(wchar_t* s3)`

Construye un objeto `_bstr_t` mediante una llamada a `SysAllocString` para crear un nuevo objeto `BSTR` y, a continuación, lo encapsula.

`_bstr_t(_variant_t& var)`

Construye un objeto `_bstr_t` a partir de un objeto `_variant_t`, para lo que primero recupera un objeto `BSTR` del objeto `VARIANT` encapsulado.

`_bstr_t(BSTR bstr, bool fCopy)`

Construye un objeto `_bstr_t` a partir de un objeto `BSTR` existente (en lugar de una cadena `wchar_t*`). Si `fCopy` es `false`, el objeto `BSTR` proporcionado se adjunta al nuevo objeto sin crear una copia con `SysAllocString`. Las funciones contenedoras usan este constructor en los encabezados de la biblioteca de tipos para encapsular y tomar la propiedad de un objeto `BSTR` devuelto por un método de interfaz.

Consulte también

[Clase _bstr_t](#)

[Clase _variant_t](#)

`_bstr_t::Assign`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Copia un valor `BSTR` en el valor `BSTR` contenido en `_bstr_t`.

Sintaxis

C++

```
void Assign(  
    BSTR s  
)
```

Parámetros

`s`

`BSTR` que se copia en el objeto `BSTR` contenido en `_bstr_t`.

Comentarios

`Assign` realiza una copia binaria de toda la longitud de `BSTR`, independientemente del contenido.

Ejemplo

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
```

FIN de Específicos de Microsoft

Consulte también

[Clase _bstr_t](#)

`_bstr_t::Attach`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Vincula un contenedor `_bstr_t` a un `BSTR`.

Sintaxis

C++

```
void Attach(  
    BSTR s  
)
```

Parámetros

`s`

`BSTR` que se va a asociar con, o asignar a, la variable `_bstr_t`.

Comentarios

Si `_bstr_t` se ha asociado previamente a otro `BSTR`, `_bstr_t` limpiará el recurso de `BSTR`, si ninguna otra variable `_bstr_t` está utilizando `BSTR`.

Ejemplo

Consulte [_bstr_t::Assign](#) para ver un ejemplo con `Attach`.

FIN de Específicos de Microsoft

Consulte también

[Clase _bstr_t](#)

`_bstr_t::copy`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Crea una copia del objeto `BSTR` encapsulado.

Sintaxis

C++

```
BSTR copy( bool fCopy = true ) const;
```

Parámetros

`fCopy`

Si `true`, `copy` copia devuelve una copia del objeto contenido `BSTR`; de lo contrario, `copy` devuelve el `BSTR` real.

Comentarios

Devuelve una copia recién asignada del objeto encapsulado `BSTR` o el propio objeto encapsulado, según el parámetro.

Ejemplo

C++

```
STDMETHODIMP CAalertMsg::get_ConnectionStr(BSTR *pVal){ // m_bsConStr is  
_bstr_t  
    *pVal = m_bsConStr.copy();  
}
```

FIN de Específicos de Microsoft

Consulte también

[Clase `_bstr_t`](#)

`_bstr_t::Detach`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Devuelve el `BSTR` contenido en `_bstr_t` y desasocia `BSTR` de `_bstr_t`.

Sintaxis

C++

```
BSTR Detach( ) throw;
```

Valor devuelto

Devuelve el `BSTR` encapsulado por `_bstr_t`.

Ejemplo

Consulte [_bstr_t::Assign](#) para ver un ejemplo en el que se usa `Detach`.

FIN de Específicos de Microsoft

Consulte también

[Clase _bstr_t](#)

`_bstr_t::GetAddress`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Libera cualquier cadena existente y devuelve la dirección de una cadena recién asignada.

Sintaxis

C++

```
BSTR* GetAddress( );
```

Valor devuelto

Puntero a `BSTR` contenido en `_bstr_t`.

Comentarios

`GetAddress` afecta a todos los objetos `_bstr_t` que comparten `BSTR`. Más de un `_bstr_t` puede compartir un `BSTR` mediante el uso del constructor de copia y `operator=`.

Ejemplo

Consulte [_bstr_t::Assign](#) para ver un ejemplo en el que se usa `GetAddress`.

FIN de Específicos de Microsoft

Consulte también

[Clase _bstr_t](#)

`_bstr_t::GetBSTR`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Señala al principio del objeto `BSTR` incluido en `_bstr_t`.

Sintaxis

C++

```
BSTR& GetBSTR();
```

Valor devuelto

Principio del objeto `BSTR` incluido en `_bstr_t`.

Comentarios

`GetBSTR` afecta a todos los objetos `_bstr_t` que comparten `BSTR`. Más de un `_bstr_t` puede compartir un `BSTR` mediante el uso del constructor de copia y `operator=`.

Ejemplo

Consulte [`_bstr_t::Assign`](#) para ver un ejemplo en el que se usa `GetBSTR`.

FIN de Específicos de Microsoft

Consulte también

[Clase `_bstr_t`](#)

`_bstr_t::length`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Devuelve el número de caracteres de `_bstr_t`, sin incluir el carácter null de terminación, del `BSTR` encapsulado.

Sintaxis

C++

```
unsigned int length( ) const throw( );
```

Comentarios

FIN de Específicos de Microsoft

Consulte también

[Clase `_bstr_t`](#)

`_bstr_t::operator =`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Asigna un nuevo valor a un objeto `_bstr_t` existente.

Sintaxis

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);
```

Parámetros

`s1`

Un objeto `_bstr_t` que se va a asignar a un objeto `_bstr_t` existente.

`s2`

Una cadena multibyte que se va a asignar a un objeto `_bstr_t` existente.

`s3`

Una cadena Unicode que se va a asignar a un objeto `_bstr_t` existente.

`var`

Un objeto `_variant_t` que se va a asignar a un objeto `_bstr_t` existente.

FIN de Específicos de Microsoft

Ejemplo

Consulte [_bstr_t::Assign](#) para obtener un ejemplo que usa `operator=`.

Consulte también

[Clase _bstr_t](#)

`_bstr_t::operator +=, _bstr_t::operator`

+

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Anexa los caracteres al final del objeto `_bstr_t` o concatena dos cadenas.

Sintaxis

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);
```

Parámetros

`s1`

Un objeto `_bstr_t`.

`s2`

Cadena multibyte.

`s3`

Cadena Unicode.

Comentarios

Estos operadores realizan la concatenación de cadenas:

- `operator+=(s1)` Anexa los caracteres en la `BSTR` encapsulada de `s1` al final de la `BSTR` encapsulada de este objeto.
- `operator+(s1)` Devuelve la nueva `_bstr_t` que se forma por la concatenación de `BSTR` de este objeto y de aquella en `s1`.
- `operator+(s2, s1)` Devuelve una `_bstr_t` que se forma por la concatenación de una `s2` de cadenas multibyte, convertida a Unicode, y la `BSTR` encapsulada en `s1`.

- `operator+(s3, s1)` Devuelve la `_bstr_t` nueva que se forma por la concatenación de una `s3` de cadena Unicode y la `BSTR` encapsulada en `s1`.

FIN de Específicos de Microsoft

Consulte también

[Clase _bstr_t](#)

`_bstr_t::operator !`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Comprueba si `BSTR` encapsulado es una cadena NULL.

Sintaxis

C++

```
bool operator!( ) const throw( );
```

Valor devuelto

Devuelve `true` si `BSTR` encapsulado es una cadena NULL, `false` si no.

FIN de Específicos de Microsoft

Consulte también

[Clase _bstr_t](#)

Operadores relacionales `_bstr_t`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Compara dos objetos `_bstr_t`.

Sintaxis

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( );
```

Comentarios

Estos operadores comparan dos objetos `_bstr_t` lexicográficamente. Los operadores devuelven `true` si las comparaciones se sostienen; si no, devuelven `false`.

FIN de Específicos de Microsoft

Consulte también

[Clase `_bstr_t`](#)

`_bstr_t::wchar_t*`, `_bstr_t::char*`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Devuelve los caracteres `BSTR` como matriz de caracteres estrechos o anchos.

Sintaxis

C++

```
operator const wchar_t*( ) const throw( );
operator wchar_t*( ) const throw( );
operator const char*( ) const;
operator char*( ) const;
```

Comentarios

Estos operadores pueden utilizarse para extraer los datos de caracteres encapsulados por el objeto `BSTR`. La asignación de un nuevo valor al puntero devuelto no modifica los datos `BSTR` originales.

FIN de Específicos de Microsoft

Consulte también

[Clase _bstr_t](#)

Clase `_com_error`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Un `_com_error` objeto representa una condición de excepción detectada por las funciones contenedoras de control de errores en los archivos de encabezado generados a partir de la biblioteca de tipos o por una de las clases de compatibilidad COM. La `_com_error` clase encapsula el `HRESULT` código de error y cualquier objeto asociado `IErrorInfo Interface`.

Construcción

| Nombre | Descripción |
|-------------------------|---|
| <code>_com_error</code> | Construye un objeto <code>_com_error</code> . |

Operadores

| Nombre | Descripción |
|-------------------------|--|
| <code>operator =</code> | Asigna un objeto <code>_com_error</code> existente a otro. |

Funciones extractoras

| Nombre | Descripción |
|------------------------|---|
| <code>Error</code> | Recupera el <code>HRESULT</code> objeto pasado al constructor . |
| <code>ErrorInfo</code> | Recupera el objeto <code>IErrorInfo</code> pasado al constructor. |
| <code>WCode</code> | Recupera el código de error de 16 bits asignado al encapsulado <code>HRESULT</code> . |

funciones `IErrorInfo`

| Nombre | Descripción |
|--------------------------|--|
| <code>Description</code> | Llama a la función <code>IErrorInfo::GetDescription</code> . |
| <code>HelpContext</code> | Llama a la función <code>IErrorInfo::GetHelpContext</code> . |

| Nombre | Descripción |
|----------|---|
| HelpFile | Llama a la función <code>IErrorInfo::GetHelpFile</code> . |
| Source | Llama a la función <code>IErrorInfo::GetSource</code> . |
| GUID | Llama a la función <code>IErrorInfo::GetGUID</code> . |

Extractor de mensajes de formato

| Nombre | Descripción |
|--------------|---|
| ErrorMessage | Recupera el mensaje de cadena para <code>HRESULT</code> almacenado en el <code>_com_error</code> objeto . |

ExepInfo.wCode a `HRESULT` los asignadores

| Nombre | Descripción |
|-----------------------------|---|
| <code>HRESULTToWCode</code> | Asigna de 32 bits <code>HRESULT</code> a 16 bits <code>wCode</code> . |
| <code>WCodeToHRESULT</code> | Asigna de 16 bits <code>wCode</code> a 32 bits <code>HRESULT</code> . |

FIN de Específicos de Microsoft

Requisitos

Encabezado:<comdef.h>

Biblioteca:`comsuppw.Lib` o `comsuppwd.Lib` (para obtener más información, consulte
`/Zc:wchar_t` (`wchar_t` es de tipo nativo))

Consulte también

[Clases de compatibilidad con COM del compilador](#)
[IErrorInfo Interfaz](#)

Funciones miembro `_com_error`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Para obtener información sobre las `_com_error` funciones miembro, vea [_com_error la clase](#).

Consulte también

[Clase _com_error](#)

`_com_error::_com_error`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Construye un objeto `_com_error`.

Sintaxis

C++

```
_com_error(
    HRESULT hr,
    IErrorInfo* perrinfo = NULL,
    bool fAddRef = false) throw();

_com_error( const _com_error& that ) throw();
```

Parámetros

`hr`

`HRESULT` Información.

`perrinfo`

Objeto `IErrorInfo`.

`fAddRef`

El valor predeterminado hace que el constructor no llame a AddRef en una interfaz que no sea `NULL` `IErrorInfo`. Este comportamiento proporciona un recuento de referencias correcto en el caso común en el que la propiedad de la interfaz se pasa al `_com_error` objeto, como:

C++

```
throw _com_error(hr, perrinfo);
```

Si no desea que el código transfiera la propiedad al `_com_error` objeto y `AddRef` es necesario desplazar el `Release` elemento en el `_com_error` destructor, construya el objeto de la siguiente manera:

C++

```
_com_error err(hr, perrinfo, true);
```

that

Objeto `_com_error` existente.

Comentarios

El primer constructor crea un nuevo objeto dado un `HRESULT` objeto y opcional `IErrorInfo`. El segundo crea una copia de un objeto existente `_com_error`.

FIN de Específicos de Microsoft

Consulte también

[Clase _com_error](#)

`_com_error::Description`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Llama a la función `IErrorInfo::GetDescription`.

Sintaxis

C++

```
_bstr_t Description() const;
```

Valor devuelto

Devuelve el resultado de `IErrorInfo::GetDescription` para el objeto `IErrorInfo` registrado dentro del objeto `_com_error`. El `BSTR` resultante se encapsula en un objeto `_bstr_t`. Si no se registra ningún `IErrorInfo`, devuelve un `_bstr_t` vacío.

Comentarios

Llama a la función `IErrorInfo::GetDescription` y recupera los `IErrorInfo` registrados dentro del objeto `_com_error`. Cualquier error que se produzca mientras se llama al método `IErrorInfo::GetDescription` se omite.

FIN de Específicos de Microsoft

Consulte también

[Clase _com_error](#)

`_com_error::Error`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Recupera el `HRESULT` objeto pasado al constructor .

Sintaxis

C++

```
HRESULT Error() const throw();
```

Valor devuelto

Elemento `HRESULT` sin formato pasado en el constructor.

Comentarios

Recupera el elemento encapsulado `HRESULT` en un `_com_error` objeto .

FIN de Específicos de Microsoft

Consulte también

[Clase _com_error](#)

`_com_error::ErrorInfo`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Recupera el objeto `IErrorInfo` pasado al constructor.

Sintaxis

C++

```
IErrorInfo * ErrorInfo( ) const throw( );
```

Valor devuelto

Elemento `IErrorInfo` sin formato pasado en el constructor.

Comentarios

Recupera el elemento encapsulado `IErrorInfo` en un `_com_error` objeto o `NULL` si no se registra ningún `IErrorInfo` elemento. El llamador debe llamar a `Release` en el objeto devuelto cuando termine de usarlo.

FIN de Específicos de Microsoft

Consulte también

[Clase _com_error](#)

`_com_error::ErrorMessage`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Recupera el mensaje de cadena para `HRESULT` almacenado en el `_com_error` objeto .

Sintaxis

C++

```
const TCHAR * ErrorMessage() const throw();
```

Valor devuelto

Devuelve el mensaje de cadena para el `HRESULT` objeto grabado dentro del `_com_error` objeto . `HRESULT` Si es un objeto asignado de 16 bits `wCode`, se devuelve un mensaje genérico "`IDispatch error #<wCode>`". Si no se encuentra ningún mensaje, se devuelve un mensaje genérico "`Unknown error #<HRESULT>`". La cadena devuelta es una cadena Unicode o multibyte, en función del estado de la `_UNICODE` macro.

Observaciones

Recupera el texto del mensaje del sistema adecuado para `HRESULT` que se registre en el `_com_error` objeto . El texto del mensaje del sistema se obtiene llamando a la función Win32 `FormatMessage` . La API asigna la `FormatMessage` cadena devuelta y se libera cuando se destruye el `_com_error` objeto.

FIN de Específicos de Microsoft

Consulte también

[Clase _com_error](#)

`_com_error::GUID`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Llama a la función `IErrorInfo::GetGUID`.

Sintaxis

C++

```
GUID GUID() const throw();
```

Valor devuelto

Devuelve el resultado de `IErrorInfo::GetGUID` para el objeto `IErrorInfo` registrado dentro del objeto `_com_error`. Si no se registra ningún objeto `IErrorInfo`, devuelve `GUID_NULL`.

Comentarios

Cualquier error que se produzca mientras se llama al método `IErrorInfo::GetGUID` se omite.

FIN de Específicos de Microsoft

Consulte también

[Clase _com_error](#)

`_com_error::HelpContext`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Llama a la función `IErrorInfo::GetHelpContext`.

Sintaxis

C++

```
DWORD HelpContext() const throw();
```

Valor devuelto

Devuelve el resultado de `IErrorInfo::GetHelpContext` para el objeto `IErrorInfo` registrado dentro del objeto `_com_error`. Si no se registra ningún objeto `IErrorInfo`, devuelve un cero.

Comentarios

Cualquier error que se produzca mientras se llama al método `IErrorInfo::GetHelpContext` se omite.

FIN de Específicos de Microsoft

Consulte también

[Clase _com_error](#)

`_com_error::HelpFile`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Llama a la función `IErrorInfo::GetHelpFile`.

Sintaxis

C++

```
_bstr_t HelpFile() const;
```

Valor devuelto

Devuelve el resultado de `IErrorInfo::GetHelpFile` para el objeto `IErrorInfo` registrado dentro del objeto `_com_error`. El BSTR resultante se encapsula en un objeto `_bstr_t`. Si no se registra ningún `IErrorInfo`, devuelve un `_bstr_t` vacío.

Comentarios

Cualquier error que se produzca mientras se llama al método `IErrorInfo::GetHelpFile` se omite.

FIN de Específicos de Microsoft

Consulte también

[Clase _com_error](#)

`_com_error::HRESULTToWCode`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Asigna de 32 bits `HRESULT` a 16 bits `wCode`.

Sintaxis

C++

```
static WORD HRESULTToWCode(  
    HRESULT hr  
) throw();
```

Parámetros

`hr`

El objeto de 32 bits `HRESULT` que se va a asignar a 16 bits `wCode`.

Valor devuelto

Se asignan de 16 bits `wCode` a partir de la clase de 32 bits `HRESULT`.

Comentarios

Para obtener más información, vea [_com_error::WCode](#).

FIN de Específicos de Microsoft

Consulte también

[_com_error::WCode](#)

[_com_error::WCodeToHRESULT](#)

[Clase _com_error](#)

`_com_error::Source`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Llama a la función `IErrorInfo::GetSource`.

Sintaxis

C++

```
_bstr_t Source() const;
```

Valor devuelto

Devuelve el resultado de `IErrorInfo::GetSource` para el objeto `IErrorInfo` registrado dentro del objeto `_com_error`. El `BSTR` resultante se encapsula en un objeto `_bstr_t`. Si no se registra ningún `IErrorInfo`, devuelve un `_bstr_t` vacío.

Comentarios

Cualquier error que se produzca mientras se llama al método `IErrorInfo::GetSource` se omite.

FIN de Específicos de Microsoft

Consulte también

[Clase _com_error](#)

`_com_error::WCode`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Recupera el código de error de 16 bits asignado al encapsulado `HRESULT`.

Sintaxis

C++

```
WORD WCode ( ) const throw();
```

Valor devuelto

`HRESULT` Si está dentro del intervalo 0x80040200 a 0x8004FFFF, el `wCode` método devuelve el `HRESULT` 0x80040200 menos; de lo contrario, devuelve cero.

Comentarios

El método `WCode` se usa para deshacer una asignación que tiene lugar en el código de compatibilidad con COM. El contenedor de una propiedad o método `dispinterface` llama a una rutina de compatibilidad que empaqueta los argumentos y llama a `IDispatch::Invoke`. Tras la devolución, si se devuelve un error `HRESULT` de , la información de `DISP_E_EXCEPTION` error se recupera de la `EXCEPINFO` estructura pasada a `IDispatch::Invoke`. El código de error puede ser un valor de 16 bits almacenado en el miembro `wCode` de la estructura `EXCEPINFO` o un valor completo de 32 bits del miembro `scode` de la estructura `EXCEPINFO`. Si se devuelve un error de 16 bits `wCode` , primero debe asignarse a un error `HRESULT` de 32 bits.

FIN de Específicos de Microsoft

Consulte también

[_com_error::HRESULTToWCode](#)

[_com_error::WCodeToHRESULT](#)

[Clase _com_error](#)

`_com_error::WCodeToHRESULT`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Asigna de 16 bits `wCode` a 32 bits `HRESULT`.

Sintaxis

C++

```
static HRESULT WCodeToHRESULT(
    WORD wCode
) throw();
```

Parámetros

`wCode`

El objeto de 16 bits `wCode` que se va a asignar a 32 bits `HRESULT`.

Valor devuelto

Asignado de 32 bits `HRESULT` a partir de la clase de 16 bits `wCode`.

Comentarios

Consulte la [WCode función miembro](#).

FIN de Específicos de Microsoft

Consulte también

[_com_error::WCode](#)

[_com_error::HRESULTToWCode](#)

[Clase _com_error](#)

Operadores de `_com_error`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Para obtener información sobre los `_com_error` operadores, vea [_com_error class](#).

Consulte también

[Clase _com_error](#)

`_com_error::operator=`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Asigna un objeto `_com_error` existente a otro.

Sintaxis

C++

```
_com_error& operator=(  
    const _com_error& that  
) throw ();
```

Parámetros

that

Un objeto `_com_error`.

FIN de Específicos de Microsoft

Consulte también

[Clase `_com_error`](#)

_com_ptr_t (Clase)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Un objeto `_com_ptr_t` encapsula un puntero de interfaz COM que se conoce como puntero "inteligente". Esta clase de plantilla administra la asignación y la desasignación de recursos con llamadas de función a las funciones miembro de `IUnknown`:

`QueryInterface`, `AddRef` y `Release`.

La definición `typedef` proporcionada por la macro `_COM_SMARTPTR_TYPEDEF` normalmente hace referencia a un puntero inteligente. Esta macro toma un nombre de interfaz y el IID, y declara una especialización de `_com_ptr_t` con el nombre de la interfaz más un sufijo de `Ptr`. Por ejemplo:

C++

```
_COM_SMARTPTR_TYPEDEF(IMyInterface, __uuidof(IMyInterface));
```

declara la especialización `IMyInterfacePtr` de `_com_ptr_t`.

Un conjunto de [plantillas de función](#), no miembros de esta clase de plantilla, admite comparaciones con un puntero inteligente a la derecha del operador de comparación.

Construcción

| Nombre | Descripción |
|----------------------------|---|
| _com_ptr_t | Construye un objeto <code>_com_ptr_t</code> . |

Operaciones de bajo nivel

| Nombre | Descripción |
|--------------------------------|--|
| AddRef | Llama a la función miembro <code>AddRef</code> de <code>IUnknown</code> en el puntero de interfaz encapsulado. |
| Adjuntar | Encapsula un puntero de interfaz sin formato del tipo de este puntero inteligente. |
| CreateInstance | Crea una nueva instancia de un objeto dado <code>CLSID</code> o <code>ProgID</code> . |

| Nombre | Descripción |
|---------------------------------|--|
| Separar | Extrae y devuelve el puntero de interfaz encapsulado. |
| GetActiveObject | Se adjunta a una instancia existente de un objeto, dado <code>CLSID</code> o <code>ProgID</code> . |
| GetInterfacePtr | Devuelve el puntero de interfaz encapsulado. |
| QueryInterface | Llama a la función miembro <code>QueryInterface</code> de <code>IUnknown</code> en el puntero de interfaz encapsulado. |
| Versión | Llama a la función miembro <code>Release</code> de <code>IUnknown</code> en el puntero de interfaz encapsulado. |

Operadores

| Nombre | Descripción |
|---|--|
| <code>operator =</code> | Asigna un nuevo valor a un objeto <code>_com_ptr_t</code> existente. |
| <code>operators ==, !=, <, >, <=, >=</code> | Compara el objeto de puntero inteligente con otro puntero inteligente, puntero de interfaz sin formato o NULL. |
| Extractores | Extrae el puntero de interfaz COM encapsulado. |

FIN de Específicos de Microsoft

Requisitos

Header: <comip.h>

Biblioteca: comsuppw.lib o comsuppwd.lib (vea [/Zc:wchar_t \(wchar_t es un tipo nativo\)](#) para obtener más información)

Consulte también

[Clases de compatibilidad con COM del compilador](#)

_com_ptr_t (Funciones miembro)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Para obtener información sobre las funciones miembro `_com_ptr_t`, consulte la [clase `_com_ptr_t`](#).

Consulte también

[_com_ptr_t \(Clase\)](#)

_com_ptr_t::_com_ptr_t

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Construye un objeto _com_ptr_t.

Sintaxis

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<_0therIID>& 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
);
```

Parámetros

pInterface

Puntero a interfaz sin formato.

fAddRef

Si es `true`, se llama a `AddRef` para incrementar el recuento de referencias del puntero a interfaz encapsulado.

cp

A objeto `_com_ptr_t`.

p

Puntero a interfaz sin formato cuyo tipo es diferente del tipo de puntero inteligente de este objeto `_com_ptr_t`.

varSrc

Un objeto `_variant_t`.

clsid

El `CLSID` de una coclase.

dwClsContext

Contexto para el código ejecutable.

lpcStr

Cadena multibyte que contiene un `CLSID` (que comienza con "{") o `ProgID`.

pOuter

El desconocido externo para [agregación](#).

FIN de Específicos de Microsoft

Consulte también

[_com_ptr_t \(Clase\)](#)

`_com_ptr_t::AddRef`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Llama a la función miembro `AddRef` de `IUnknown` en el puntero de interfaz encapsulado.

Sintaxis

C++

```
void AddRef( );
```

Comentarios

Llama a `IUnknown::AddRef` en el puntero de interfaz encapsulado y provoca un error `E_POINTER` si el puntero es NULL.

FIN de Específicos de Microsoft

Consulte también

[_com_ptr_t \(Clase\)](#)

_com_ptr_t::Attach

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Encapsula un puntero de interfaz sin formato del tipo de este puntero inteligente.

Sintaxis

C++

```
void Attach( Interface* pInterface ) throw( );
void Attach( Interface* pInterface, bool fAddRef ) throw( );
```

Parámetros

pInterface

Puntero a interfaz sin formato.

fAddRef

Si es `true`, se llama a `AddRef`. Si es `false`, el objeto `_com_ptr_t` toma la propiedad del puntero de interfaz sin formato sin llamar a `AddRef`.

Comentarios

- No se llama a `Attach(pInterface) AddRef`. La propiedad de la interfaz se pasa a este objeto `_com_ptr_t`. Se llama a `Release` para disminuir el recuento de referencias del puntero previamente encapsulado.
- `Attach(pInterface,fAddRef)` Si `fAddRef` es `true`, se llama a `AddRef` para incrementar el recuento de referencias del puntero de interfaz encapsulado. Si `fAddRef` es `false`, este objeto `_com_ptr_t` toma la propiedad del puntero de interfaz sin formato sin llamar a `AddRef`. Se llama a `Release` para disminuir el recuento de referencias del puntero previamente encapsulado.

FIN de Específicos de Microsoft

Consulte también

`_com_ptr_t` (Clase)

_com_ptr_t::CreateInstance

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Crea una nueva instancia de un objeto dado `CLSID` o `ProgID`.

Sintaxis

```
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( );
```

Parámetros

rclsid

El `CLSID` de un objeto.

clsidString

Cadena Unicode que contiene un `CLSID` (que comienza con "{") o `ProgID`.

clsidStringA

Cadena multibyte, en la página de códigos ANSI, que contiene un `CLSID` (que comienza con "{") o `ProgID`.

dwClsContext

Contexto para el código ejecutable.

pOuter

El desconocido externo para [agregación](#).

Comentarios

Estas funciones de miembro llaman a `CoCreateInstance` para crear un nuevo objeto CM y, a continuación, consultas para el tipo de interfaz de este puntero inteligente. El puntero resultante se encapsula dentro de este objeto `_com_ptr_t`. Se llama a `Release` para disminuir el recuento de referencias para el puntero previamente encapsulado. Esta rutina devuelve el HRESULT para indicar si la operación se ha realizado de forma correcta o no.

- **CreateInstance** (*rclsid,dwClContext*) Crea una nueva instancia en ejecución de un objeto, `CLSID` dado un.
- **CreateInstance** (*clsidString,dwClContext*) Crea una nueva instancia en ejecución de un objeto, dada una cadena Unicode que contiene `CLSID` (a partir de " {`ProgID`}") o.
- **CreateInstance** (*clsidStringA,dwClContext*) Crea una nueva instancia en ejecución de un objeto a partir de una cadena de caracteres multibyte `CLSID` que contiene (a partir de " {`ProgID`}") o. Llama a `MultiByteToWideChar`, que supone que la cadena está en la página de códigos ANSI en lugar de una página de códigos OEM.

FIN de Específicos de Microsoft

Consulte también

[_com_ptr_t \(Clase\)](#)

`_com_ptr_t::Detach`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Extrae y devuelve el puntero de interfaz encapsulado.

Sintaxis

```
Interface* Detach( ) throw( );
```

Comentarios

Extrae y devuelve el puntero de interfaz encapsulado y, a continuación, borra el almacenamiento del puntero encapsulado como NULL. Esto quita el puntero de interfaz de la encapsulación. Es decisión suya llamar a `Release` en el puntero de interfaz devuelto.

FIN de Específicos de Microsoft

Consulte también

[_com_ptr_t \(Clase\)](#)

_com_ptr_t::GetActiveObject

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Se adjunta a una instancia existente de un objeto, dado un `CLSID` o `ProgID`.

Sintaxis

```
HRESULT GetActiveObject(
    const CLSID& rclsid
) throw( );
HRESULT GetActiveObject(
    LPCWSTR clsidString
) throw( );
HRESULT GetActiveObject(
    LPCSTR clsidStringA
) throw( );
```

Parámetros

rclsid

El `CLSID` de un objeto.

clsidString

Cadena Unicode que contiene un `CLSID` (que comienza con "{") o `ProgID`.

clsidStringA

Cadena multibyte, en la página de códigos ANSI, que contiene un `CLSID` (que comienza con "{") o `ProgID`.

Comentarios

Estas funciones miembro llaman a `GetActiveObject` para recuperar un puntero a un objeto actual que se ha registrado con OLE y, después, consultan el tipo de interfaz de este puntero inteligente. El puntero resultante se encapsula dentro de este objeto `_com_ptr_t`. Se llama a `Release` para disminuir el recuento de referencias para el puntero previamente encapsulado. Esta rutina devuelve el `HRESULT` para indicar si la operación se ha realizado de forma correcta o no.

- **GetActiveObject(`rclsid`)** Se adjunta a una instancia existente de un objeto, dado un `CLSID`.
- **GetActiveObject(`clsidString`)** Se adjunta a una instancia existente de un objeto, dada una cadena Unicode que contiene un `CLSID` (que comienza con "{") o un `ProgID`.
- **GetActiveObject(`clsidStringA`)** Se adjunta a una instancia existente de un objeto, dada una cadena Unicode que contiene un `CLSID` (que comienza con "{") o un `ProgID`. Llama a [MultiByteToWideChar](#), que supone que la cadena está en la página de códigos ANSI en lugar de una página de códigos OEM.

FIN de Específicos de Microsoft

Consulte también

[_com_ptr_t \(Clase\)](#)

_com_ptr_t::GetInterfacePtr

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Devuelve el puntero de interfaz encapsulado.

Sintaxis

```
Interface* GetInterfacePtr( ) const throw( );
Interface*& GetInterfacePtr() throw();
```

Comentarios

Devuelve el puntero de interfaz encapsulado, que puede ser NULL.

FIN de Específicos de Microsoft

Consulte también

[_com_ptr_t \(Clase\)](#)

_com_ptr_t::QueryInterface

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Llama a la función miembro `QueryInterface` de `IUnknown` en el puntero de interfaz encapsulado.

Sintaxis

```
template<typename _InterfaceType> HRESULT QueryInterface (
    const IID& iid,
    _InterfaceType*& p
) throw( );
template<typename _InterfaceType> HRESULT QueryInterface (
    const IID& iid,
    _InterfaceType** p
) throw( );
```

Parámetros

iid

`IID` de un puntero de interfaz.

p

Puntero de interfaz sin formato.

Comentarios

Llama a `IUnknown::QueryInterface` en el puntero de interfaz encapsulado con el `IID` especificado y devuelve el puntero de interfaz sin formato resultante en *p*. Esta rutina devuelve el `HRESULT` para indicar si la operación se ha realizado de forma correcta o no.

FIN de Específicos de Microsoft

Consulte también

[_com_ptr_t \(Clase\)](#)

`_com_ptr_t::Release`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Llama a la función miembro `Release` de `IUnknown` en el puntero de interfaz encapsulado.

Sintaxis

C++

```
void Release( );
```

Comentarios

Llama a `IUnknown::Release` en el puntero encapsulado de interfaz y genera un error `E_POINTER` si este puntero de interfaz es NULL.

FIN de Específicos de Microsoft

Consulte también

[_com_ptr_t \(Clase\)](#)

_com_ptr_t (Operadores)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Para obtener información sobre los operadores `_com_ptr_t`, consulte [Clase _com_ptr_t](#).

Consulte también

[_com_ptr_t \(Clase\)](#)

`_com_ptr_t::operator =`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Asigna un nuevo valor a un objeto `_com_ptr_t` existente.

Sintaxis

```
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 );
```

Comentarios

Asigna un puntero de interfaz a este objeto `_com_ptr_t`.

FIN de Específicos de Microsoft

Consulte también

[_com_ptr_t \(Clase\)](#)

`_com_ptr_t` (Operadores relacionales)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Compara el objeto de puntero inteligente con otro puntero inteligente, puntero de interfaz sin formato o NULL.

Sintaxis

```
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 );
```

Comentarios

Compara el objeto de puntero inteligente a otro puntero inteligente, puntero de interfaz sin formato o NULL. A excepción de las comprobaciones de puntero nulo, estos operadores consultan primero ambos punteros para determinar si son `IUnknown` y comparan los resultados.

FIN de Específicos de Microsoft

Consulte también

[_com_ptr_t \(Clase\)](#)

`_com_ptr_t` Extractores

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específico de Microsoft

Extrae el puntero de interfaz COM encapsulado.

Sintaxis

C++

```
operator Interface*( ) const throw( );
operator Interface&( ) const;
Interface& operator*( ) const;
Interface* operator->( ) const;
Interface** operator&( ) throw( );
operator bool( ) const throw( );
```

Comentarios

- `operator Interface*` devuelve el puntero de interfaz encapsulado, que puede ser `NULL`.
- `operator Interface&` devuelve una referencia al puntero de interfaz encapsulado y emite un error si el puntero es `NULL`.
- `operator*` permite que un objeto de puntero inteligente actúe como si fuera la interfaz encapsulada real cuando se desreferencia.
- `operator->` permite que un objeto de puntero inteligente actúe como si fuera la interfaz encapsulada real cuando se desreferencia.
- `operator&` libera cualquier puntero de interfaz encapsulado, lo reemplaza por `NULL` y devuelve la dirección del puntero encapsulado. Este operador permite pasar el puntero inteligente por dirección a una función que tenga un parámetro `out` a través del cual devuelve un puntero de interfaz.
- `operator bool` permite usar un objeto de puntero inteligente en una expresión condicional. Este operador devuelve `true` si el puntero no es `NULL`.

(!) Nota

Dado que `operator bool` no se declara como `explicit`, `_com_ptr_t` se puede convertir implícitamente en `bool`, que es convertible en cualquier tipo de escalar. Esto puede tener consecuencias inesperadas en el código. Habilite la [Advertencia del compilador \(nivel 4\) C4800](#) para evitar el uso no intencionado de esta conversión.

Consulte también

[_com_ptr_t \(clase\)](#)

Plantillas de funciones relacionales

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Sintaxis

```
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
);
```

Parámetros

i

Puntero a interfaz sin formato.

p

Un puntero inteligente.

Comentarios

Estas plantillas de función permiten realizar la comparación con un puntero inteligente a la derecha del operador de comparación. No son funciones miembro de `_com_ptr_t`.

FIN de Específicos de Microsoft

Consulte también

[_com_ptr_t \(Clase\)](#)

_variant_t (Clase)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Un objeto `_variant_t` encapsula el tipo de datos `VARIANT`. La clase administra la asignación y desasignación de recursos y realiza llamadas de función a `VariantInit` y `VariantClear` según corresponda.

Construcción

| Nombre | Descripción |
|-------------------------|---|
| <code>_variant_t</code> | Construye un objeto <code>_variant_t</code> . |

Operaciones

| Nombre | Descripción |
|-------------------------|---|
| <code>Adjuntar</code> | Adjunta un objeto <code>VARIANT</code> al objeto <code>_variant_t</code> . |
| <code>Borrar</code> | Borra el objeto <code>VARIANT</code> encapsulado. |
| <code>ChangeType</code> | Cambia el tipo de objeto <code>_variant_t</code> al <code>VARTYPE</code> indicado. |
| <code>Separar</code> | Desasocia el objeto encapsulado <code>VARIANT</code> de este objeto <code>_variant_t</code> . |
| <code>SetString</code> | Asigna una cadena a este objeto <code>_variant_t</code> . |

Operadores

| Nombre | Descripción |
|------------------------------|---|
| <code>Operador =</code> | Asigna un nuevo valor a un objeto <code>_variant_t</code> existente. |
| <code>operador ==, !=</code> | Compara dos objetos <code>_variant_t</code> para ver si son iguales o no. |
| <code>Extractores</code> | Extraen datos del objeto <code>VARIANT</code> encapsulado. |

FIN de Específicos de Microsoft

Requisitos

Encabezado:<comutil.h>

Biblioteca: comsuppw.lib o comsuppwd.lib (vea [/Zc:wchar_t \(wchar_t es tipo nativo\)](#) para obtener más información)

Consulte también

[Clases de compatibilidad con COM del compilador](#)

_variant_t (Funciones miembro)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Para obtener información sobre las funciones miembro `_variant_t`, consulta [_variant_t Class](#).

Consulte también

[_variant_t \(Clase\)](#)

`_variant_t::_variant_t`

Artículo • 03/03/2023 • Tiempo de lectura: 4 minutos

Específicos de Microsoft

Construye un objeto `_variant_t`.

Sintaxis

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();
```

Parámetros

`varSrc`

Un objeto `VARIANT` que se va a copiar en el nuevo objeto `_variant_t`.

`pVarSrc`

Puntero a un objeto `VARIANT` que se va a copiar en el nuevo objeto `_variant_t`.

`var_t_Src`

Un objeto `_variant_t` que se va a copiar en el nuevo objeto `_variant_t`.

`fCopy`

Si es `false`, el objeto `VARIANT` proporcionado se adjunta al nuevo objeto `_variant_t` sin crear una copia con `VariantCopy`.

`ISrc, sSrc`

Un valor entero que se va a copiar en el nuevo objeto `_variant_t`.

`vtSrc`

`VARTYPE` para el nuevo objeto `_variant_t`.

`fLtSrc, dblSrc`

Un valor numérico que se va a copiar en el nuevo objeto `_variant_t`.

`cySrc`

Un objeto `cy` que se va a copiar en el nuevo objeto `_variant_t`.

`bstrSrc`

Un objeto `_bstr_t` que se va a copiar en el nuevo objeto `_variant_t`.

`strSrc, wstrSrc`

Una cadena que se va a copiar en el nuevo objeto `_variant_t`.

`bSrc`

Un valor `bool` que se va a copiar en el nuevo objeto `_variant_t`.

`pIUnknownSrc`

Puntero de interfaz COM a un objeto VT_UNKNOWN que se va a encapsular en el nuevo objeto `_variant_t`.

`pDispSrc`

Puntero de interfaz COM a un objeto VT_DISPATCH que se va a encapsular en el nuevo objeto `_variant_t`.

`decSrc`

Un valor `DECIMAL` que se va a copiar en el nuevo objeto `_variant_t`.

`bSrc`

Un valor `BYTE` que se va a copiar en el nuevo objeto `_variant_t`.

`cSrc`

Un valor `char` que se va a copiar en el nuevo objeto `_variant_t`.

`usSrc`

Un valor `unsigned short` que se va a copiar en el nuevo objeto `_variant_t`.

`ulSrc`

Un valor `unsigned long` que se va a copiar en el nuevo objeto `_variant_t`.

`iSrc`

Un valor `int` que se va a copiar en el nuevo objeto `_variant_t`.

`uiSrc`

Un valor `unsigned int` que se va a copiar en el nuevo objeto `_variant_t`.

`i8Src`

Un valor `__int64` que se va a copiar en el nuevo objeto `_variant_t`.

`ui8Src`

Un valor `unsigned __int64` que se va a copiar en el nuevo objeto `_variant_t`.

Comentarios

- `_variant_t()` construye un objeto vacío `_variant_t`, `VT_EMPTY`.

- `_variant_t(VARIANT& varSrc)` construye un objeto `_variant_t` a partir de una copia del objeto `VARIANT`. El tipo variant se conserva.
- `_variant_t(VARIANT* pVarSrc)` construye un objeto `_variant_t` a partir de una copia del objeto `VARIANT`. El tipo variant se conserva.
- `_variant_t(_variant_t& var_t_Src)` construye un objeto `_variant_t` desde otro objeto `_variant_t`. El tipo variant se conserva.
- `_variant_t(VARIANT& varSrc, bool fCopy)` construye un objeto `_variant_t` a partir de un objeto `VARIANT` existente. Si `fCopy` es `false`, el objeto `VARIANT` se asocia al nuevo objeto sin crear una copia.
- `_variant_t(short sSrc, VARTYPE vtSrc = VT_I2)` construye un objeto `_variant_t` de tipo `VT_I2` o `VT_BOOL` a partir de un valor entero `short`. Cualquier otro `VARTYPE` producirá un error `E_INVALIDARG`.
- `_variant_t(long lSrc, VARTYPE vtSrc = VT_I4)` construye un objeto `_variant_t` de tipo `VT_I4`, `VT_BOOL` o `VT_ERROR` a partir de un valor entero `long`. Cualquier otro `VARTYPE` producirá un error `E_INVALIDARG`.
- `_variant_t(float fltSrc)` construye un objeto `_variant_t` de tipo `VT_R4` a partir de un valor `float`.
- `_variant_t(double dblSrc, VARTYPE vtSrc = VT_R8)` construye un objeto `_variant_t` de tipo `VT_R8` o `VT_DATE` a partir de un valor `double`. Cualquier otro `VARTYPE` producirá un error `E_INVALIDARG`.
- `_variant_t(CY& cySrc)` construye un objeto `_variant_t` de tipo `VT_CY` a partir de un objeto `CY`.
- `*_variant_t(_bstr_t& bstrSrc)` construye un objeto `_variant_t` de tipo `VT_BSTR` a partir de un objeto `_bstr_t`. Se asigna un nuevo `BSTR`.
- `_variant_t(wchar_t* wstrSrc)` construye un objeto `_variant_t` de tipo `VT_BSTR` a partir de un cadena Unicode. Se asigna un nuevo `BSTR`.
- `_variant_t(char* strSrc)` construye un objeto `_variant_t` de tipo `VT_BSTR` a partir de una cadena. Se asigna un nuevo `BSTR`.
- `_variant_t(bool bSrc)` construye un objeto `_variant_t` de tipo `VT_BOOL` a partir de un valor `bool`.

- `_variant_t(IUnknown* pIUnknownSrc, bool fAddRef = true)` construye un objeto `_variant_t` de tipo `VT_UNKNOWN` a partir de un puntero de interfaz COM. Si `fAddRef` es `true`, entonces se llama a `AddRef` en el puntero de interfaz proporcionado para que coincida con la llamada a `Release` que se producirá cuando se destruya el objeto `_variant_t`. Es decisión suya llamar a `Release` en el puntero de interfaz proporcionado. Si `fAddRef` es `false`, este constructor toma la propiedad del puntero de interfaz proporcionado; no debe llamar a `Release` en el puntero de interfaz proporcionado.
- `_variant_t(IDispatch* pDispSrc, bool fAddRef = true)` construye un objeto `_variant_t` de tipo `VT_DISPATCH` a partir de un puntero de interfaz COM. Si `fAddRef` es `true`, entonces se llama a `AddRef` en el puntero de interfaz proporcionado para que coincida con la llamada a `Release` que se producirá cuando se destruya el objeto `_variant_t`. Es decisión suya llamar a `Release` en el puntero de interfaz proporcionado. Si `fAddRef` es `false`, este constructor toma la propiedad del puntero de interfaz proporcionado; no debe llamar a `Release` en el puntero de interfaz proporcionado.
- `_variant_t(DECIMAL& decSrc)` construye un objeto `_variant_t` de tipo `VT_DECIMAL` a partir de un valor `DECIMAL`.
- `_variant_t(BYTE bSrc)` construye un objeto `_variant_t` de tipo `VT_UI1` a partir de un valor `BYTE`.

FIN de Específicos de Microsoft

Consulte también

[Clase _variant_t](#)

_variant_t::Attach

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Adjunta un objeto `VARIANT` al objeto `_variant_t`.

Sintaxis

C++

```
void Attach(VARIANT& varSrc);
```

Parámetros

varSrc

Objeto `VARIANT` que se adjuntará a este objeto `_variant_t`.

Comentarios

Toma la propiedad de `VARIANT` encapsulándolo. Esta función miembro libera cualquier `VARIANT` encapsulado existente, copia el `VARIANT` proporcionado y establece su `VARTYPE` en `VT_EMPTY` para garantizar que solo el destructor `_variant_t` puede liberar sus recursos.

FIN de Específicos de Microsoft

Consulte también

[_variant_t \(Clase\)](#)

`_variant_t::Clear`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Borra el objeto `VARIANT` encapsulado.

Sintaxis

C++

```
void Clear( );
```

Comentarios

Llama a `VariantClear` en el objeto `VARIANT` encapsulado.

FIN de Específicos de Microsoft

Consulte también

[_variant_t \(Clase\)](#)

`_variant_t::ChangeType`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Cambia el tipo del objeto `_variant_t` al `VARTYPE` indicado.

Sintaxis

C++

```
void ChangeType(
    VARTYPE vartype,
    const _variant_t* pSrc = NULL
);
```

Parámetros

vartype

`VARTYPE` para este objeto `_variant_t`.

pSrc

Un puntero al objeto `_variant_t` que se va a convertir. Si este valor es `NULL`, la conversión se realiza en contexto.

Comentarios

Esta función miembro convierte un objeto `_variant_t` en el `VARTYPE` indicado. Si *pSrc* es `NULL`, la conversión se realiza en contexto; de lo contrario, este objeto `_variant_t` se copia de *pSrc* y después se convierte.

FIN de Específicos de Microsoft

Consulte también

[_variant_t \(Clase\)](#)

`_variant_t::Detach`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Desasocia el objeto encapsulado `VARIANT` de este `_variant_t` objeto.

Sintaxis

```
VARIANT Detach( );
```

Valor devuelto

`VARIANT` encapsulado.

Comentarios

Extrae y devuelve el objeto `VARIANT` encapsulado y después borra este objeto `_variant_t` sin destruirlo. Esta función miembro quita el objeto `VARIANT` de la encapsulación y establece el `VARTYPE` de este objeto `_variant_t` en `VT_EMPTY`. Decida si va a liberar el objeto `VARIANT` devuelto llamando a la función [VariantClear](#).

FIN de Específicos de Microsoft

Consulte también

[_variant_t \(Clase\)](#)

`_variant_t::SetString`

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Asigna una cadena a este objeto `_variant_t`.

Sintaxis

C++

```
void SetString(const char* pSrc);
```

Parámetros

pSrc

Puntero a la cadena de caracteres.

Comentarios

Convierte una cadena de caracteres ANSI en una cadena `BSTR` Unicode y la asigna a este objeto `_variant_t`.

FIN de Específicos de Microsoft

Consulte también

[_variant_t \(Clase\)](#)

_variant_t (Operadores)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Para obtener información sobre los operadores `_variant_t`, consulte [Clase _variant_t](#).

Consulte también

[_variant_t \(Clase\)](#)

`_variant_t::operator=`

Artículo • 03/03/2023 • Tiempo de lectura: 3 minutos

Asigna un nuevo valor a una instancia `_variant_t`.

La clase `_variant_t` y su miembro `operator=` son específicos de Microsoft.

Sintaxis

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 );
```

Parámetros

`varSrc`

Referencia a un `VARIANT` desde el que se va a copiar el contenido y el tipo `VT_*`.

`pVarSrc`

Puntero a un `VARIANT` desde el que se va a copiar el contenido y el tipo `VT_*`.

`var_t_Src`

Referencia a un `_variant_t` desde el que se va a copiar el contenido y el tipo `VT_*`.

`sSrc`

Valor `short` entero que se va a copiar. Tipo dado `VT_BOOL` si `*this` es del tipo `VT_BOOL`. De lo contrario, se le asigna el tipo `VT_I2`.

`lSrc`

Un valor `long` entero que se va a copiar. Tipo dado `VT_BOOL` si `*this` es del tipo `VT_BOOL`. Tipo dado `VT_ERROR` si `*this` es del tipo `VT_ERROR`. De lo contrario, se ha especificado el tipo `VT_I4`.

`fltSrc`

Valor `float` numérico que se va a copiar. Dado el tipo `VT_R4`.

`dblSrc`

Valor `double` numérico que se va a copiar. Tipo dado `VT_DATE` si `this` es del tipo `VT_DATE`. De lo contrario, se ha especificado el tipo `VT_R8`.

`cySrc`

Objeto `CY` que se va a copiar. Dado el tipo `VT_CY`.

`bstrSrc`

Objeto `BSTR` que se va a copiar. Dado el tipo `VT_BSTR`.

`wstrSrc`

Cadena Unicode que se va a copiar, almacenada como `BSTR` y dado el tipo `VT_BSTR`.

`strSrc`

Cadena multibyte que se va a copiar, almacenada como `BSTR` y dado el tipo `VT_BSTR`.

`pDispSrc`

Puntero `IDispatch` que se va a copiar con una llamada a `AddRef`. Dado el tipo `VT_DISPATCH`.

`bSrc`

Valor `bool` que se va a copiar. Dado el tipo `VT_BOOL`.

`pSrc`

Puntero `IUnknown` que se va a copiar con una llamada a `AddRef`. Dado el tipo `VT_UNKNOWN`.

`decSrc`

Objeto `DECIMAL` que se va a copiar. Dado el tipo `VT_DECIMAL`.

`byteSrc`

Valor `BYTE` que se va a copiar. Dado el tipo `VT_UI1`.

`cSrc`

Valor `char` que se va a copiar. Dado el tipo `VT_I1`.

`usSrc`

Valor `unsigned short` que se va a copiar. Dado el tipo `VT_UI2`.

`ulSrc`

Valor `unsigned long` que se va a copiar. Dado el tipo `VT_UI4`.

`iSrc`

Valor `int` que se va a copiar. Dado el tipo `VT_INT`.

`uiSrc`

Valor `unsigned int` que se va a copiar. Dado el tipo `VT_UINT`.

`i8Src`

Valor `__int64` o `long long` que se va a copiar. Dado el tipo `VT_I8`.

`ui8Src`

Valor `unsigned __int64` o `unsigned long long` que se va a copiar. Dado el tipo `VT_UI8`.

Comentarios

El operador `operator=` de asignación borra cualquier valor existente, que elimina los tipos de objeto o llama a `Release` para los tipos `IDispatch*` y `IUnknown*`. A continuación, copia un nuevo valor en el objeto `_variant_t`. Cambia el tipo `_variant_t` para que coincida con el valor asignado, excepto como se indica en los argumentos `short`, `long` y `double`. Los tipos de valor se copian directamente. Un argumento de referencia o un puntero `VARIANT` o `_variant_t` copia el contenido y el tipo del objeto asignado. Otros argumentos de tipo de puntero o referencia crean una copia del objeto asignado. El operador de asignación llama `AddRef` a argumentos `IDispatch*` y `IUnknown*`.

`operator=` invoca a `_com_raise_error` si se produce un error.

`operator=` devuelve una referencia al objeto `_variant_t` actualizado.

Consulte también

Clase _variant_t

_variant_t (Operadores relacionales)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Compara dos objetos `_variant_t` para ver si son iguales o no.

Sintaxis

```
bool operator==(  
    const VARIANT& varSrc) const;  
bool operator==(  
    const VARIANT* pSrc) const;  
bool operator!=(  
    const VARIANT& varSrc) const;  
bool operator!=(  
    const VARIANT* pSrc) const;
```

Parámetros

varSrc

`VARIANT` que se va a comparar con el objeto `_variant_t`.

pSrc

Puntero a `VARIANT` que se va a comparar con el objeto `_variant_t`.

Valor devuelto

Devuelve `true` si la comparación se mantiene, si no `false`.

Comentarios

Compara un objeto `_variant_t` con `VARIANT`, para comprobar la igualdad o desigualdad.

FIN de Específicos de Microsoft

Consulte también

[_variant_t \(Clase\)](#)

`_variant_t` (Extractores)

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

Específicos de Microsoft

Extraen datos del objeto `VARIANT` encapsulado.

Sintaxis

```
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;
```

Comentarios

Extrae datos sin formato del objeto `VARIANT` encapsulado. Si `VARIANT` todavía no es del tipo adecuado, se usa `VariantChangeType` para intentar la conversión y, si no se produce, se genera un error:

- **operator short()** Extrae un valor entero `short`.
- **operator long()** Extrae un valor entero `long`.
- **operator float()** Extrae un valor numérico `float`.
- **operator double()** Extrae un valor entero `double`.

- **operator CY()** Extrae un objeto `CY`.
- **operator bool()** Extrae un valor `bool`.
- **operator DECIMAL()** Extrae un valor `DECIMAL`.
- **operator BYTE()** Extrae un valor `BYTE`.
- **operator _bstr_t()** Extrae una cadena, que se encapsula en un objeto `_bstr_t`.
- **operator IDispatch*()** Extrae un puntero dispinterface de un objeto `VARIANT` encapsulado. Se llama a `AddRef` en el puntero resultante, por lo que debe decidir si es conveniente llamar a `Release` para liberarlo.
- **operator IUnknown*()** Extrae un puntero de interfaz COM de un objeto `VARIANT` encapsulado. Se llama a `AddRef` en el puntero resultante, por lo que debe decidir si es conveniente llamar a `Release` para liberarlo.

FIN de Específicos de Microsoft

Consulte también

[_variant_t \(Clase\)](#)

Extensiones de Microsoft

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

asm-statement:

```
_asm assembly-instruction ; opt
_asm { assembly-instruction-list } ; opt
```

assembly-instruction-list:

```
assembly-instruction ; opt
assembly-instruction ; assembly-instruction-list ; opt
```

ms-modifier-list:

```
ms-modifier ms-modifier-list opt
```

ms-modifier:

```
_cdecl
_fastcall
_stdcall
_syscall (reservado para futuras implementaciones)
_oldcall (reservado para futuras implementaciones)
_unaligned (reservado para futuras implementaciones)
```

based-modifier

based-modifier:

```
_based ( based-type )
```

based-type:

```
name
```

Comportamiento no estándar

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

En las secciones siguientes se enumeran algunos de los lugares en los que la implementación de Microsoft de C++ no se ajusta al estándar de C++. Los números de sección que se indican a continuación se refieren a los números de sección del estándar de C++ 11 (ISO/IEC 14882:2011(E)).

La lista de límites del compilador que difieren de los definidos en el estándar de C++ se proporciona en [Límites del compilador](#).

Tipos de valor devueltos de covariante

Las clases base virtuales no se admiten como tipos de valor devueltos de covariante cuando la función virtual tiene un número variable de argumentos. Esto no se ajusta a la sección 10.3, párrafo 7 de la especificación ISO de C++ 11. El ejemplo siguiente no se compila; genera el error del compilador [C2688](#):

```
C++

// CovariantReturn.cpp
class A
{
    virtual A* f(int c, ...);    // remove ...
};

class B : virtual A
{
    B* f(int c, ...);    // C2688 remove ...
};
```

Enlazar nombres no dependientes en plantillas

En estos momentos, el compilador de Microsoft C++ no admite nombres no dependientes al analizar una plantilla inicialmente. Esto no se ajusta a la sección 14.6.3 de la especificación ISO de C++ 11. Esto puede hacer que se vean las sobrecargas declaradas después de la plantilla (pero antes de que se creen instancias de la plantilla).

```
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)

```

Especificadores de excepciones de funciones

Los especificadores de excepciones de funciones distintos de `throw()` se analizan pero no se usan. Esto no se ajusta a la sección 15.4 de la especificación ISO C++ 11. Por ejemplo:

C++

```

void f() throw(int); // parsed but not used
void g() throw();    // parsed and used

```

Para obtener más información sobre las especificaciones de excepciones, vea [Especificaciones de excepciones](#).

char_traits::eof()

El estándar de C++ indica que `char_traits::eof` no debe corresponder a un valor `char_type` válido. El compilador de Microsoft C++ exige esta restricción para el tipo `char`, pero no para el tipo `wchar_t`. Esto no se ajusta al requisito de la tabla 62 de la sección 12.1.1 de la especificación ISO de C++ 11. En el ejemplo siguiente se muestra este comportamiento.

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;
}
```

Ubicación de almacenamiento de objetos

El estándar de C++ (sección 1.8, párrafo 6) requiere que los objetos de C++ completos tengan ubicaciones de almacenamiento únicas. Sin embargo, con Microsoft C++ hay casos en los que tipos sin miembros de datos compartirán una ubicación de almacenamiento con otros tipos mientras dure el objeto.

Límites del compilador

Artículo • 03/03/2023 • Tiempo de lectura: 2 minutos

El estándar de C++ recomienda límites para varias construcciones del lenguaje. La siguiente es una lista de casos en los que el compilador de Microsoft C++ no implementa los límites recomendados. El primer número es el límite que se establece en el estándar ISO C++ 11 (INCITS/ISO/IEC 14882-2011[2012], Anexo B) y el segundo número es el límite implementado por el compilador de Microsoft C++:

- Niveles de anidamiento de declaraciones compuestas, estructuras de control de iteración y estructuras de control de selección: estándar de C++: 256, compilador de Microsoft C++: depende de la combinación de declaraciones anidadas, pero generalmente entre 100 y 110.
- Parámetros en una definición de macro: estándar de C++: 256, compilador de Microsoft C++: 127.
- Argumentos en una invocación de macro: C++ estándar: 256, compilador de Microsoft C++ 127.
- Caracteres en un literal de cadena de caracteres o literal de cadena ancha (después de la concatenación): estándar de C++: 65536, compilador de Microsoft C++: 65535 caracteres de un solo byte, incluido el terminador NULL, y 32767 caracteres de doble byte, incluido el terminador NULL.
- Niveles de definiciones anidadas de clase, estructura o unión en un único estándar `struct-declaration-list` de C++: 256, compilador de Microsoft C++: 16.
- Inicializadores de miembros en una definición de constructor: estándar de C++: 6144, compilador de Microsoft C++: al menos 6144.
- Calificaciones de alcance de un identificador: estándar de C++: 256, compilador de Microsoft C++: 127.
- Se anidaron `extern` especificaciones estándar de C++: 1024, compilador de Microsoft C++: 9 (sin contar la especificación implícita `extern` en el ámbito global, o 10, si cuenta la especificación implícita `extern` en el ámbito global).
- Argumentos de plantilla en una declaración de plantilla: estándar de C++: 1024, compilador de Microsoft C++: 2046.

Consulte también

Comportamiento no estándar

Referencia del preprocesador de C/C++

Artículo • 26/09/2022 • Tiempo de lectura: 2 minutos

La *Referencia del preprocesador de C/C++* explica el preprocesador tal como se implementa en Microsoft C/C++. El preprocesador realiza operaciones preliminares en archivos de C y C++ antes de pasarlos al compilador. Puede usar el preprocesador para realizar las siguientes acciones de manera condicional: compilar código, insertar archivos, especificar mensajes de error en tiempo de compilación y aplicar reglas específicas del equipo a secciones del código.

En Visual Studio 2019, la opción del compilador [/Zc:preprocessor](#) proporciona un preprocesador C11 y C17 totalmente compatible. Este es el valor predeterminado cuando se usa la marca [/std:c11](#) del compilador o [/std:c17](#).

En esta sección

[Preprocesador](#)

Proporciona información general sobre los preprocesadores tradicionales y nuevos conformes.

[Directivas de preprocesador](#)

Describe las directivas, utilizadas normalmente para que los programas de origen sean fáciles de modificar y de compilar en diferentes entornos de ejecución.

[Operadores de preprocesador](#)

Describe los cuatro operadores específicos del preprocesador usados en el contexto de la directiva `#define`.

[Macros predefinidas](#)

Describe las macros predefinidas especificadas por los estándares de C y C++ y por Microsoft C++.

[Pragmas](#)

Describe las directivas pragma, que proporcionan un método para que cada compilador ofrezca características específicas del equipo o del sistema operativo a la vez que conserva la compatibilidad total con los lenguajes C y C++.

Secciones relacionadas

[Referencia del lenguaje C++](#)

Proporciona material de referencia para la implementación de Microsoft del lenguaje

C++.

[Referencia del lenguaje C](#)

Proporciona material de referencia para la implementación de Microsoft del lenguaje C.

[Referencia de compilación de C/C++](#)

Proporciona vínculos a temas que describen las opciones del compilador y el vinculador.

[Proyectos de Visual Studio: C++](#)

Describe la interfaz de usuario de Visual Studio que permite especificar los directorios en los que el sistema de proyectos buscará los archivos para el proyecto de C++.

Referencia de la biblioteca estándar (SLTL) de C++

Artículo • 16/11/2022 • Tiempo de lectura: 2 minutos

Un programa de C++ puede llamar a un gran número de funciones desde esta implementación conforme de la biblioteca estándar de C++. Estas funciones realizan servicios como entrada y salida, y proporcionan implementaciones eficaces de operaciones utilizadas con frecuencia.

Para obtener más información sobre la vinculación con el archivo en tiempo de ejecución `.lib` de Microsoft Visual C++ adecuado, vea [Archivos .lib de tiempo de ejecución de C \(CRT\) y de la biblioteca estándar de C++ \(STL\)](#).

ⓘ Nota

La implementación de Microsoft de la biblioteca estándar de C++ se conoce a menudo como *STL* o *biblioteca de plantillas estándar*. Aunque la *biblioteca estándar de C++* es el nombre oficial de la biblioteca tal como se define en ISO 14882, debido al uso popular de "STL" y "biblioteca de plantillas estándar" en los motores de búsqueda, en ocasiones usamos esos nombres para simplificar la búsqueda de nuestra documentación.

Desde una perspectiva histórica, "STL" se refería originalmente a la biblioteca de plantillas estándar escrita por Alexander Stepanov. Las partes de esa biblioteca se estandarizaron en la biblioteca estándar de C++, junto con la biblioteca en tiempo de ejecución de C ISO, las partes de la biblioteca Boost y otra funcionalidad. A veces se usa "STL" para hacer referencia a las partes de los contenedores y algoritmos de la biblioteca estándar de C++ adaptada de la STL de Stepanov. En esta documentación, la biblioteca de plantillas estándar (STL) hace referencia a la biblioteca estándar de C++ en su conjunto.

En esta sección

[Información general de la biblioteca estándar de C++](#) Proporciona información general sobre la implementación de Microsoft de la biblioteca estándar de C++.

[Programación de iostream](#) Proporciona información general de la programación de `iostream`.

[Referencia de archivos de encabezado](#) Proporciona vínculos a temas de referencia en los que se describen los archivos de encabezado de la biblioteca estándar de C++, con ejemplos de código.