

Contents

[Documentación de lenguaje C++](#)

[Referencia del lenguaje C++](#)

[Referencia del lenguaje C++](#)

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

[Convenciones léxicas](#)

[Convenciones léxicas](#)

[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](#)

[Conceptos básicos](#)

[Conceptos básicos](#)

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

[Ámbito](#)

[Archivos de encabezado](#)

[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](#)

[Tipos de valor](#)

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

[Conversiones estándar](#)

Tipos integrados

Tipos integrados

Intervalos de tipo de datos

nullptr

void

bool

False

True

char, wchar_t, char16_t, char32_t

_int8, _int16, _int32, _int64

_m64

_m128

_m128d

_m128i

_ptr32, _ptr64

Límites numéricos

Límites numéricos

Límites de enteros

Límites flotantes

Declaraciones y definiciones

Declaraciones y definiciones

Clases de almacenamiento

auto

const

constexpr

extern

Inicializadores

Alias y definiciones de tipo

declaración using

volatile

decltype

Atributos

Operadores integrados, precedencia y asociatividad

Operadores integrados, precedencia y asociatividad

Operador alignof

_uuidof (Operador)

Operadores de adición: + y -

Operador address-of: &

Operadores de asignación

Operador AND bit a bit: &

Operador OR exclusivo bit a bit: ^

Operador OR inclusivo bit a bit: |

Operador de conversión: ()

Operador coma: ,

Operador condicional: ?:

delete (Operador)

Operadores de igualdad: == y !=

Operador de conversión explícita de tipos: ()

Operador de llamada de función: ()

Operador de direccionamiento indirecto: *

Operadores de desplazamiento a la izquierda y a la derecha (>> y <<)

Operador AND lógico: &&

Operador de negación lógico: !

Operador OR lógico: ||

Operadores de acceso a miembros: . y ->

Operadores de multiplicación y el operador de módulo

new (Operador)

Operador de complemento de uno: ~

Operadores de puntero a miembro: .* y ->*

Operadores de incremento y decremento postfijos: ++ y --

Operadores de incremento y decremento prefijos: ++ y --

Operadores relationales: <, >, <= y >=

Operador de resolución de ámbito: ::

sizeof (Operador)

[Operador de subíndice:](#)

[typeid \(Operador\)](#)

[Operadores unarios más y de negación: + y -](#)

[Expresiones](#)

[Expresiones](#)

[Tipos de expresiones](#)

[Tipos de expresiones](#)

[Expresiones primarias](#)

[Puntos suspensivos y plantillas variádicas](#)

[Expresiones de postfijo](#)

[Expresiones con operadores unarios](#)

[Expresiones con operadores binarios](#)

[Expresiones constantes](#)

[Semántica de las expresiones](#)

[Conversión](#)

[Conversión](#)

[Operadores de conversión](#)

[Operadores de conversión](#)

[dynamic_cast \(Operador\)](#)

[bad_cast \(Excepción\)](#)

[static_cast \(Operador\)](#)

[const_cast \(Operador\)](#)

[reinterpret_cast \(Operador\)](#)

[Información de tipos en tiempo de ejecución \(RTTI\)](#)

[Información de tipos en tiempo de ejecución \(RTTI\)](#)

[bad_typeid \(Excepción\)](#)

[type_info \(Clase\)](#)

[Instrucciones](#)

[Instrucciones](#)

[Información general sobre las instrucciones de C++](#)

[Instrucciones con etiqueta](#)

[Expression \(Instrucción\)](#)

- [Expression \(Instrucción\)](#)
- [NULL \(Instrucción\)](#)
- [Instrucciones compuestas \(Bloques\)](#)
 - [Instrucciones de selección](#)
 - [Instrucciones de selección](#)
 - [Instrucción if-else](#)
 - [_if_exists \(Instrucción\)](#)
 - [_if_not_exists \(Instrucción\)](#)
 - [switch \(Instrucción\)](#)
 - [Instrucciones de iteración](#)
 - [Instrucciones de iteración](#)
 - [while \(Instrucción\)](#)
 - [do-while \(Instrucción\)](#)
 - [for \(Instrucción\)](#)
 - [Instrucción for basada en intervalos](#)
 - [Instrucciones de salto](#)
 - [Instrucciones de salto](#)
 - [break \(Instrucción\)](#)
 - [continue \(Instrucción\)](#)
 - [return \(Instrucción\)](#)
 - [goto \(Instrucción\)](#)
 - [Transferencias del control](#)
- [Espacios de nombres](#)
- [Enumeraciones](#)
- [Uniones](#)
- [Funciones](#)
 - [Funciones](#)
 - [Funciones con listas de argumentos de variable](#)
 - [Sobrecarga de funciones](#)
 - [Funciones establecidas como valor predeterminado y eliminadas explícitamente](#)
 - [Búsqueda de nombres dependientes de argumentos \(Koenig\) en las funciones](#)
 - [Argumentos predeterminados](#)

Funciones insertadas

Sobrecarga de operadores

Sobrecarga de operadores

Reglas generales para la sobrecarga de operadores

Sobrecargar operadores unarios

Sobrecargar operadores unarios

Sobrecarga de operadores de incremento y decremento

Operadores binarios

Asignación

Llamada a función

Subíndices

Acceso a miembros

Clases y estructuras

Clases y estructuras

clase

struct

Información general sobre miembros de clase

Control de acceso a miembros

Control de acceso a miembros

friend

private

protected

public

Inicialización de llaves

Duración de objetos y administración de recursos (RAII)

Expresión Pimpl para encapsulación en tiempo de compilación

Portabilidad en los límites de ABI

Constructores

Constructores

Constructores de copia y operadores de asignación de copia

Constructores de movimiento y operadores de asignación de movimiento

Constructores de delegación

Destructores

 Información general de Funciones miembro

 Información general de Funciones miembro

 virtual (Especificador)

 override (Especificador)

 final (Especificador)

Herencia

 Herencia

 Funciones virtuales

 Herencia única

 Clases base

 Varias clases base

 Invalidaciones explícitas

 Clases abstractas

 Resumen de reglas de ámbito

 Palabras clave de herencia

 virtual

 __super

 __interface

 Funciones miembro especiales

 Miembros estáticos

 Clases C++ como tipos de valor

 Conversiones de tipos definidos por el usuario

 Miembros de datos mutables

 Declaraciones de clase anidadas

 Tipos de clase anónima

 Punteros a miembros

 this (Puntero)

 Campos de bits

Expresiones lambda en C++

 Expresiones lambda en C++

 Sintaxis de la expresión lambda

Ejemplos de expresiones lambda

Expresiones lambda constexpr

Matrices

Referencias

Referencias

Declarador de referencia a un valor L: &

Declarador de referencia a un valor R: &&

Argumentos de función de tipo de referencia

Valores devueltos de función de tipo de referencia

Referencias a punteros

Punteros

Punteros

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

Pimpl para encapsulación en tiempo de compilación

Control de excepciones en C++

Control de excepciones en C++

Procedimientos recomendados modernos de C++

Cómo: Diseñar para la seguridad de las excepciones

Cómo: Interfaz entre código excepcional y no excepcional

Instrucciones try, throw y catch

Cómo se evalúan los bloques Catch

Excepciones y desenredo de la pila

Especificaciones de excepciones (throw)

noexcept

Excepciones de C++ no controladas

[Mezclar excepciones de C \(estructuradas\) y de C++](#)

[Mezclar excepciones de C \(estructuradas\) y de C++](#)

[Usar setjmp-longjmp](#)

[Controlar excepciones estructuradas en C++](#)

[Control de excepciones estructurado \(C/C++\)](#)

[Control de excepciones estructurado \(C/C++\)](#)

[Escribir un controlador de excepciones](#)

[Escribir un controlador de excepciones](#)

[try-except \(Instrucción\)](#)

[Escribir un filtro de excepción](#)

[Generar excepciones de software](#)

[Excepciones de hardware](#)

[Restricciones de los controladores de excepciones](#)

[Escribir un controlador de finalización](#)

[Escribir un controlador de finalización](#)

[try-finally \(Instrucción\)](#)

[Limpiar recursos](#)

[Resumen sobre los intervalos de control de excepciones Resumen](#)

[Restricciones de los controladores de finalización](#)

[Transportar excepciones entre subprocessos](#)

[Aserción y mensajes proporcionados por el usuario](#)

[Aserción y mensajes proporcionados por el usuario](#)

[static_assert](#)

[Módulos](#)

[Información general de los módulos en C++](#)

[módulo, importar, exportar](#)

[Plantillas](#)

[Plantillas](#)

[typename](#)

[Plantillas de clase](#)

[Plantillas de función](#)

[Plantillas de función](#)

- Crear instancias de plantillas de función
 - Creación de instancias explícita
 - Especialización explícita de las plantillas de función
 - Ordenación parcial de plantillas de función
 - Plantillas de función miembro
- Especialización de plantilla
 - Plantillas y resolución de nombres
 - Plantillas y resolución de nombres
 - Resolución de nombres para los tipos dependientes
 - Resolución de nombres declarados localmente
 - Resolución de sobrecarga de llamadas de plantilla de función
 - Organización de código fuente (plantillas de C++)
- Control de eventos
 - Control de eventos
 - `_event`
 - `_hook`
 - `_raise`
 - `_unhook`
 - Control de eventos en C++ nativo
 - Control de eventos en COM
- Modificadores específicos de Microsoft
 - Modificadores específicos de Microsoft
 - Direccionamiento con base
 - Direccionamiento con base
 - `_based` (Gramática)
 - Punteros con base
 - Convenciones de llamada
 - Convenciones de llamada
 - Paso de argumentos y convenciones de nomenclatura
 - Paso de argumentos y convenciones de nomenclatura
 - `_cdecl`
 - `_clrcall`

`_stdcall`

`_fastcall`

`_thiscall`

`_vectorcall`

Ejemplo de llamada: Llamada y prototipo de función

Ejemplo de llamada: Llamada y prototipo de función

Resultados del ejemplo de llamada

Llamadas de función naked

Llamadas de función naked

Reglas y limitaciones de las funciones naked

Consideraciones para escribir código de prólogo/epílogo

Coprocesador de punto flotante y convenciones de llamada

Convenciones de llamada obsoletas

`restrict` (C++ AMP)

`tile_static` (Palabra clave)

`_declspec`

`_declspec`

`align`

`allocate`

`allocator`

`appdomain`

`code_seg (_declspec)`

`deprecated`

`dllexport, dllimport`

`dllexport, dllimport`

Definiciones y declaraciones

Definir funciones insertadas de C++ con `dllexport` y `dllimport`

Reglas generales y limitaciones

Utilizar `dllimport` y `dllexport` en las clases de C++

`jit intrinsic`

`naked`

`noalias`

`noinline`
`noreturn`
`no_SANITIZE_ADDRESS`
`nothrow`
`novtable`
`proceso`
`propiedad`
`restrict`
`safebuffers`
`selectany`
`spectre`
`thread`
`uuid`
`_restrict`
`_SPTR, _UPTR`
`_unaligned`
`_w64`
`_func_`

Compatibilidad con COM del compilador
Compatibilidad con COM del compilador
Funciones globales COM de compilador
Funciones globales COM de compilador
`_com_raise_error`
`ConvertStringToBSTR`
`ConvertBSTRToString`
`_set_com_error_handler`

Clases de compatibilidad con COM del compilador
Clases de compatibilidad con COM del compilador
`_bstr_t (Clase)`
`_bstr_t (clase)`
`_bstr_t::_bstr_t`
`_bstr_t::Assign`

_bstr_t::Attach
_bstr_t::copy
_bstr_t::Detach
_bstr_t::GetAddress
_bstr_t::GetBSTR
_bstr_t::length
_bstr_t::operator =
_bstr_t::operator +=, +
_bstr_t::operator !=
_bstr_t (Operadores)
_bstr_t::wchar_t *, _bstr_t::char*
_com_error (clase)
_com_error (Clase)
_com_error (Funciones miembro)
_com_error (Funciones miembro)
_com_error::_com_error
_com_error::Description
_com_error::Error
_com_error::ErrorInfo
_com_error::ErrorMessage
_com_error::GUID
_com_error::HelpContext
_com_error::HelpFile
_com_error::HRESULTToWCode
_com_error::Source
_com_error::WCode
_com_error::WCodeToHRESULT
_com_error (Operadores)
_com_error (Operadores)
_com_error::operator =
_com_ptr_t (Clase)
_com_ptr_t (clase)

_com_ptr_t (Funciones miembro)
_com_ptr_t (Funciones miembro)
_com_ptr_t::_com_ptr_t
_com_ptr_t::AddRef
_com_ptr_t::Attach
_com_ptr_t::CreateInstance
_com_ptr_t::Detach
_com_ptr_t::GetActiveObject
_com_ptr_t::GetInterfacePtr
_com_ptr_t::QueryInterface
_com_ptr_t::Release
_com_ptr_t (Operadores)
_com_ptr_t (Operadores)
_com_ptr_t::operator =
_com_ptr_t (Operadores relacionales)
_com_ptr_t (Extractores)

Plantillas de funciones relacionales

_variant_t (Clase)
_variant_t (clase)
_variant_t (Funciones miembro)
_variant_t (Funciones miembro)
_variant_t::_variant_t
_variant_t::Attach
_variant_t::Clear
_variant_t::ChangeType
_variant_t::Detach
_variant_t::SetString
_variant_t (Operadores)
_variant_t (Operadores)
_variant_t::operator =
_variant_t (Operadores relacionales)
_variant_t (Extractores)

[Extensiones de Microsoft](#)

[Comportamiento no estándar](#)

[Límites del compilador](#)

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

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

Referencia del lenguaje C++

06/03/2021 • 5 minutes to read • [Edit Online](#)

Esta referencia 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 la norma internacional 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++, vea [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 que se integran en el compilador de C++ y sus intervalos de valores.

[Conversiones estándar](#)

Conversiones de tipos entre los 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, funciones miembro especiales, miembros de datos, campos de bits, `this` puntero, 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, `virtual` funciones, varias clases base, clases **abstractas**, reglas de ámbito.

También, las `__super` `__interface` palabras clave y.

Control de acceso a miembros

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

Mensajes de aserión y User-Supplied

`#error` Directive, la `static_assert` palabra clave, la `assert` macro.

Templates (Plantillas [C++])

Especificaciones de plantilla, plantillas de función, plantillas de clase, `typename` palabras clave, plantillas frente a macros, plantillas y punteros inteligentes.

Control de eventos

Declaración de eventos y controladores de eventos.

Modificadores específicos de Microsoft

Modificadores específicos de Microsoft C++. Direccionamiento de memoria, convenciones de llamada, `naked` funciones, atributos extendidos de clase de almacenamiento (`__declspec`), `__w64` .

Ensamblador alineado

Usar el lenguaje de ensamblado y C++ en `__asm` bloques.

Compatibilidad con COM del compilador

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

Extensiones de Microsoft

Extensiones de Microsoft a C++.

Comportamiento no estándar

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

Bienvenido de nuevo a 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 diversas bibliotecas de Microsoft C++.

Vea también

[Referencia del lenguaje C](#)

Aquí está otra vez C++: C++ moderno

02/11/2020 • 17 minutes to read • [Edit Online](#)

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. Con C++, puede crear una amplia gama de aplicaciones: juegos, controladores de dispositivos y software científico de alto rendimiento, programas incrustados y aplicaciones cliente de Windows. 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. Los antiguos recursos de programación de C están a su disposición siempre que los necesite, pero con el código C++ moderno debería necesitarlos 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 `unique_ptr` encapsula las llamadas a `new` y `delete`. Cuando un objeto `widget` sale del ámbito, se invoca el destructor `unique_ptr`, el cual liberará la memoria asignada para la matriz.

```

#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 asignar memoria de 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 inefficiencias que podrían transferirse a estructuras de datos personalizadas. En lugar de matrices sin formato, use `vector` como un contenedor secuencial en 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.

```

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 utilice 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.

```
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(), 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:

```
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 \(C++\)](#).

```

#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:

```

#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`.

```

#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 `for_each` invocará en los elementos del vector:

```

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 los errores. 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 subprocesos.

`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++](#)

[Tabla de conformidad del lenguaje Microsoft C++](#)

Convenciones léxicas

25/03/2020 • 2 minutes to read • [Edit Online](#)

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:

- [Tokens y juegos de 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 C++ cómo se analizan los archivos de código fuente, vea [fases de traducción](#).

Consulte también

[Referencia del lenguaje C++](#)

[Unidades de traducción y vinculación](#)

Juegos de tokens y caracteres

06/03/2021 • 7 minutes to read • [Edit Online](#)

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 un *espacio en blanco*, que puede 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 básico de caracteres de código fuente* 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 básico de caracteres de código fuente* consta de 96 caracteres que se pueden usar en los 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 básico de caracteres de código fuente. MSVC también permite usar un conjunto de caracteres adicional en los archivos de código fuente, en función de la codificación del archivo. De forma predeterminada, Visual Studio almacena archivos de código fuente mediante la página de códigos predeterminada. Cuando se guardan archivos de código fuente mediante una página de códigos específica de la configuración regional o una página de códigos Unicode, MSVC le permite usar cualquiera de los caracteres de esa página de códigos en el código fuente, excepto los códigos de control no permitidos explícitamente en el juego básico de caracteres de código fuente. 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 secuencias de caracteres que no se puedan traducir en 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, consulte [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 0xFFFF. 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 un carácter en forma de nombre de carácter universal y forma literal indistintamente. Por ejemplo, se puede declarar un identificador con formato de nombre de carácter universal y usarlo en el formato de literal:

```
auto \u30AD = 42; // \u30AD is 'ヰ'  
if (\u30AD == 42) return true; // \u30AD and \u30AD 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 los caracteres de control que representan la alerta, el retroceso, el retorno de carro y el carácter nulo. El juego de caracteres de ejecución tiene una representación específica de la configuración regional.

Comentarios (C++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 los comentarios en las pruebas para que algunas líneas de código queden inactivas. sin embargo, las `#if` / `#endif` directivas de preprocesador funcionan mejor para esto, ya que puede rodear el código que contiene comentarios, pero no puede 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.

Consulta también

[Convenciones léxicas](#)

Identificadores (C++)

06/03/2021 • 6 minutes to read • [Edit Online](#)

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 intervalos permitidos 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.

```
// 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 (__) en 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.

Consulta también

[Convenciones léxicas](#)

Palabras clave (C++)

09/03/2021 • 6 minutes to read • [Edit Online](#)

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 subrayados iniciales y nombres especificados para C++/CX y C++/CLI son extensiones de Microsoft.

Palabras clave de C++ estándar

alignas
alignof
and^b
and_eq^b
asm^{un}
auto
bitand^b
bitor^b
bool
break
case
catch
char
char8_t^c
char16_t
char32_t
class
compl^b
concept^c
const
const_cast
consteval^c
constexpr

constinit^c
continue
co_await^c
co_return^c
co_yield^c
decltype
default
delete
do
double
dynamic_cast
else
enum

```
explicit
export c
extern
false
float
for
friend
goto
if
inline

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

static_cast
struct
switch
template
this
thread_local
throw
true
try
typedef
typeid
typename
union
unsigned
using relativa
```

^a una palabra clave específica de Microsoft `__asm` reemplaza la `asm` Sintaxis de C++. `asm` está reservado para la compatibilidad con otras implementaciones de C++, pero no se implementa. Se usa `__asm` para el ensamblado alineado en destinos x86. Microsoft C++ no es compatible con el ensamblado alineado para otros destinos.

^b el operador extendido sinónimos son palabras clave cuando `/permissive-` se especifica o `/Za` (deshabilita extensiones de lenguaje . No son palabras clave cuando se habilitan las extensiones de Microsoft.

^c se admite cuando `/std:c++latest` se especifica.

Palabras clave de C++ específicas de Microsoft

En C++, los identificadores que contienen dos subrayados consecutivos se reservan para las implementaciones del compilador. La Convención de Microsoft es preceder a las palabras clave específicas de Microsoft con un carácter de subrayado doble. 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, puede deshabilitar las extensiones de Microsoft especificando la `/permissive-` opción o `/Za` (deshabilitar extensiones de 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 compatibilidad 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 de subrayado doble. La `__cdecl` palabra clave está disponible sin subrayado inicial.

La `__asm` palabra clave reemplaza la sintaxis de C++ `asm` . `asm` está reservado para la compatibilidad con otras implementaciones de C++, pero no se implementa. Utilice `__asm` .

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

<code>__alignof</code>	e
<code>__asm</code>	e
<code>__assume</code>	e
<code>__based</code>	e
<code>__cdecl</code>	e
<code>__declspec</code>	e
<code>__event</code>	
<code>__except</code>	e
<code>__fastcall</code>	e
<code>__finally</code>	e
<code>__forceinline</code>	e
<code>__hook</code>	d
<code>__if_exists</code>	
<code>__if_not_exists</code>	
<code>__inline</code>	e
<code>__int16</code>	e
<code>__int32</code>	e
<code>__int64</code>	e
<code>__int8</code>	e
<code>__interface</code>	
<code>__leave</code>	e
<code>__m128</code>	
<code>__m128d</code>	

```
__m128i  
__m64  
__multiple_inheritance e  
__ptr32 e  
__ptr64 e:  
__raise  
__restrict e  
__single_inheritance e:  
__sptr e:  
__stdcall e  
  
__super  
__thiscall  
__unaligned e  
__unhook d  
__uptr e  
__uuidof e  
__vectorcall e  
__virtual_inheritance e  
__w64 e  
__wchar_t
```

función intrínseca ^d utilizada en el control de eventos.

^e para mantener la compatibilidad con versiones anteriores, estas palabras clave están disponibles con dos guiones bajos iniciales y un único carácter de subrayado inicial cuando se habilitan las extensiones de Microsoft (valor predeterminado).

Palabras clave de Microsoft en modificadores `_declspec`

Estos identificadores son atributos extendidos para el `_declspec` modificador. 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_cast` f
`__value` f
`abstract` g
`array` g
`as_friend`
`delegate` g
`enum class`
`enum struct`
`event` g

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

`new` g
`property` g
`ref class`
`ref struct`
`safecast`
`sealed` g
`typeid`
`value class` g
`value struct` g

^f aplicable solo 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](#).

⁹ aplicable a C++/CLI.

Vea también

[Convenciones léxicas](#)

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

Signos de puntuación (C++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 4 de la [traducción](#).

Consulta también

[Convenciones léxicas](#)

Literales numéricos, booleanos y de puntero

06/03/2021 • 9 minutes to read • [Edit Online](#)

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

```
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. Se hace anexando prefijos o sufijos al literal. Por ejemplo, el prefijo `0x` indica al compilador que interprete el número que lo sigue como un valor hexadecimal, por ejemplo `0x35`. El `ULL` sufijo indica al compilador que trate el valor como un `unsigned long long` tipo, 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. Puede especificar literales enteros 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 el uso de un sufijo.

Si no existe ningún prefijo o sufijo, el compilador proporcionará un tipo de valor literal entero `int` (32 bits), si el valor se ajustará; de lo contrario, le proporcionará el tipo `long long` (64 bits).

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

```
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:

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

Para especificar un literal entero hexadecimal, comience la especificación con `0x` o `0X` (el caso de "x" no importa), seguido de una secuencia de dígitos en el intervalo `0` a través de `9` y `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:

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

Para especificar un tipo sin signo, use el `u` sufijo o. Para especificar un tipo Long, use el `l` sufijo 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 portátil. Por ejemplo:

```
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 ubicación en números mayores con el fin de facilitar la lectura de los usuarios. Los separadores no afectan a la compilación.

```
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 contener exponentes.

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

```
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:

```
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 predeterminado `double`. Mediante el uso de los sufijos `f` u or `l` `F` o `L` (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 del mismo tipo. Por ejemplo, puede tener funciones sobrecargadas como

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

y

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

booleanos, literales

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

Literal de puntero (C++11)

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

```
auto x = 0B0001101 ; // 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:

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

En el ejemplo anterior, se recomienda usar una constante con nombre que contenga un significado claro, por ejemplo "MAXIMUM_ERROR_THRESHOLD". Y si los usuarios finales ven el valor devuelto "Success", es posible que sea mejor usar una constante de cadena con nombre. Puede mantener constantes de cadena en una única ubicación de un archivo que se pueda localizar en otros idiomas. El uso de constantes con nombre ayuda tanto a usted como a otros a comprender el propósito del código.

Consulte también

[Convenciones léxicas](#)

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

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

Literales de cadena y carácter (C++)

02/11/2020 • 33 minutes to read • [Edit Online](#)

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 puede crear `std::string` literales sin tener que realizar pasos adicionales de construcción o conversión.

```
#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'А'; // char16_t
    auto c4 = U'А'; // char32_t

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

    // String literals
    auto s0 = "hello"; // const char*
    auto s1 = u8"hello"; // const char*, encoded as UTF-8
    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*, encoded as UTF-8
    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
    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*, encoded as UTF-8
    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 `R`, `u8R`, `LR`, `uR` los prefijos „y `UR` para los

equivalentes de versión sin formato de estas codificaciones. Para crear valores temporales o estáticos `std::string`, puede usar literales de cadena o literales de cadena sin formato con un `s` sufijo. 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 de páginas de códigos extendidas en el código fuente, vea [juegos de caracteres](#).

Literales de carácter

Un *literal de carácter* está compuesto por un carácter de constante. Se representa mediante el carácter rodeado de 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 carácter UTF-16 de tipo `char16_t`, por ejemplo `u'a'`
- Literales de carácter UTF-32 de tipo `char32_t`, por ejemplo `U'a'`

El carácter utilizado para un literal de carácter puede ser cualquier carácter, a excepción de la barra diagonal inversa de caracteres reservados (), la comilla \ simple (') 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 carácter se codifican de manera diferente según su prefijo.

- Un literal de carácter sin prefijo es un literal de carácter ordinario. El valor de un literal de carácter ordinario que contiene un carácter único, 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 ordinario que contiene más de un carácter, una secuencia de escape o un nombre de carácter universal es un *literal multicarácter*. Un literal multicarácter o un literal de carácter ordinario 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 `L` prefijo es un literal de caracteres anchos. El valor de un literal de carácter ancho que contiene un carácter único, una secuencia de escape o un 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 carácter no tenga ninguna representación en el conjunto 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 carácter universal está definido por la implementación. Para MSVC, consulte la sección [específica de Microsoft](#) más adelante.
- Un literal de carácter que comienza con el `u8` prefijo es un literal de carácter UTF-8. El valor de un literal de carácter UTF-8 que contiene un carácter único, una secuencia de escape o un nombre de carácter universal tiene un valor igual a su valor de punto de código ISO 10646 si se puede representar mediante una única unidad de código UTF-8 (que corresponde a los controles C0 y el bloque Unicode latín básico). Si el valor no se puede representar mediante una única unidad de código UTF-8, el programa tiene un formato incorrecto. Un literal de carácter UTF-8 que contiene más de un carácter, una secuencia de escape o un nombre de carácter universal tiene un formato incorrecto.
- Un literal de carácter que comienza con el `u` prefijo es un literal de carácter UTF-16. El valor de un literal

de carácter UTF-16 que contiene un carácter único, una secuencia de escape o un nombre de carácter universal tiene un valor igual a su valor de punto de código ISO 10646 si se puede representar mediante una única unidad de código UTF-16 (correspondiente al plano multilingüe básico). Si el valor no se puede representar mediante una única unidad de código UTF-16, el programa tiene un formato incorrecto. Un literal de carácter UTF-16 que contiene más de un carácter, una secuencia de escape o un nombre de carácter universal tiene un formato incorrecto.

- Un literal de carácter que comienza con el prefijo es un literal de carácter UTF-32. El valor de un literal de carácter UTF-32 que contiene un carácter único, una secuencia de escape o un nombre de carácter universal tiene un valor igual a su valor de punto de código ISO 10646. Un literal de carácter UTF-32 que contiene más de un carácter, una secuencia de escape o un nombre de carácter universal tiene un formato incorrecto.

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	\c
Carácter nulo	\0
avance de página	\f
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 del tercer dígito. El valor octal más alto posible es .

Una secuencia de escape hexadecimal es una barra diagonal inversa seguida del carácter x seguido de una secuencia de uno o más dígitos hexadecimales. Los ceros a la izquierda se ignoran. En un literal de carácter normal o U8, el valor hexadecimal más alto 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 que usan literales de carácter ordinarios. La misma sintaxis de secuencia de escape es válida para los demás tipos de literales de carácter.

```
#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 desea 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 multitarácter estrecho, el compilador convierte el carácter o la secuencia de caracteres entre comillas simples en valores de 8 bits dentro de un entero de 32 bits. Varios caracteres del literal rellenan los bytes correspondientes según sea necesario de orden superior a orden inferior. A continuación, el compilador convierte el entero al tipo de destino siguiendo las reglas habituales. Por ejemplo, para crear un `char` valor, el compilador toma el byte de orden inferior. Para crear un `wchar_t` `char16_t` valor o, el compilador toma 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.

```
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 tres dígitos seguida de los dígitos subsiguientes como caracteres en un literal de varios caracteres, que puede proporcionar resultados sorprendentes. Por ejemplo:

```
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 los caracteres siguientes en un

literal de multicarácter. Se genera una advertencia C4125 si el primer carácter que no es octal es un dígito decimal. Por ejemplo:

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

Una secuencia de escape octal con un valor mayor que `\377` produce 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 multicarácter 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".

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

Si un literal de carácter ancho con el prefijo `L` contiene una secuencia de varios caracteres, el valor se toma del primer carácter y el compilador genera la advertencia C4066. Los caracteres siguientes se omiten, a diferencia del comportamiento del literal multicarácter normal equivalente.

```
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** finaliza 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.

```
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 en el intervalo de puntos de código suplemente D800-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 subjuego de caracteres mediante sus propios nombres de carácter universal. También permitía algunos nombres de carácter universal que, en efecto, no representaban ningún carácter Unicode válido. 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 de 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.

```
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 de 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.

```
const char* str1 = u8"Hello World";  
const char* str2 = u8"\U0001F607 is O:-)";
```

Literales de cadena anchos

Un literal de cadena ancho es una matriz terminada en NULL de una constante `wchar_t` que tiene el prefijo ' L ' y 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.

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

char16_t y char32_t (C++11)

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

```
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](#).

```
// 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_utf8 = u8R"(An unescaped \ character)";
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 `\()` 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:

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

Pero un delimitador lo resuelve:

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

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

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

literales `STD:: String` (C++ 14)

`std::string` los literales son implementaciones de biblioteca estándar de literales definidos por el usuario (vea más abajo) que se representan como `"xyz"s` (con un `s` sufijo). Este tipo de literal de cadena produce un objeto temporal de tipo `std::string`, `std::wstring`, `std::u32string` o `std::u16string`, dependiendo del prefijo especificado. Cuando no se usa ningún prefijo, como antes, `std::string` se genera un. `L"xyz"s` produce una `std::wstring`. `u"xyz"s` genera un `STD::u16string` `U"xyz"s` genera `STD::u32string`.

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

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

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

`std::string` los literales se definen en el espacio de nombres del `std::literals::string_literals` `<string>` archivo de encabezado. Dado que `std::literals::string_literals`, y `std::literals` se declaran como `espacios`

de nombres alineados, `std::literals::string_literals` se trata automáticamente como si pertenecía directamente al espacio de nombres `std`.

Tamaño de literales de cadena

En el caso de las `char*` cadenas ANSI y otras codificaciones de un solo byte (pero no UTF-8), el tamaño (en bytes) de un literal de cadena es el número de caracteres más 1 para el carácter nulo de terminación. En el caso de todos los demás tipos de cadena, el tamaño no está estrechamente relacionado con el número de caracteres. UTF-8 usa hasta cuatro `char` elementos para codificar algunas *unidades de código*, `char16_t` o `wchar_t` codificadas como UTF-16 pueden usar dos elementos (para un total de cuatro bytes) para codificar una sola unidad de *código*. En este ejemplo se muestra el tamaño, en bytes, de un literal de cadena ancho:

```
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 al tamaño del elemento de cadena: un byte en una `char*` `char8_t*` cadena o, dos bytes en las `wchar_t*` cadenas o `char16_t*`, y cuatro bytes en las `char32_t*` cadenas.

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.

Modificar literales de cadena

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

Espífico de Microsoft

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

```
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 no const al establecer la opción del compilador `/zc:strictStrings` ([deshabilitar conversión de tipo de literal de cadena](#)) . Es recomendable para el código portable que cumple los estándares. También se recomienda usar la `auto` palabra clave para declarar punteros inicializados con literales de cadena, ya que se resuelve en el tipo correcto (Const). Por ejemplo, en este ejemplo de código se detecta un intento de escribir en un literal de cadena en tiempo de compilación:

```
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 `/GF` opción del compilador.

La sección [espcífica de Microsoft](#) finaliza aquí.

Concatenación de literales de cadena adyacentes

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

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

es idéntica a esta declaración:

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

y a esta declaración:

```
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:

```
"\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 usar una de estas secuencias de escape:

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

`std::string` los literales, porque son `std::string` tipos, se pueden concatenar con el `+` operador que se define para los `basic_string` tipos. También pueden concatenarse de la misma manera que literales de cadena adyacentes. En ambos casos, la codificación de cadena y el sufijo deben coincidir:

```
auto x1 = "hello" " " " world"; // OK  
auto x2 = U"hello" " " L"world"; // C2308: disagree on prefix  
auto x3 = u8"hello" " " s u8"world"s; // OK, agree on prefixes and suffixes  
auto x4 = 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, un nombre de carácter universal que representa un carácter extendido no se puede codificar en una cadena estrecha mediante la página de códigos ANSI, pero se puede codificar en cadenas estrechas en algunas páginas de códigos de varios bytes o en cadenas UTF-8 o en una cadena de caracteres anchos. En C++ 11, la compatibilidad con Unicode se extiende mediante los `char16_t*` `char32_t*` tipos de cadena y:

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

// UTF-16 (on Windows) encoded WINKING FACE (U+1F609)
const wchar_t*  s2 = L"\u0001F609 is ;-)";

// UTF-8 encoded SMILING FACE WITH HALO (U+1F607)
const char*     s3 = u8"\u0001F607 is O:-)";

// UTF-16 encoded SMILING FACE WITH OPEN MOUTH (U+1F603)
const char16_t* s4 = u"\u0001F603 is :-D";

// UTF-32 encoded SMILING FACE WITH SUNGLASSES (U+1F60E)
const char32_t* s5 = U"\u0001F60E is B-)";
```

Consulte también

[Juegos de caracteres](#)

[Literales numéricos, booleanos y de puntero](#)

[Literales definidos por el usuario](#)

Literales definidos por el usuario

02/11/2020 • 9 minutes to read • [Edit Online](#)

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 accesos directos sintácticos para las expresiones comunes y aumentar la seguridad de tipos. Por ejemplo, supongamos que tiene una `Distance` clase. Puede definir un literal para kilómetros y otro para millas, y animar al usuario a que sea explícito sobre las unidades de medida escribiendo: `auto d = 42.0_km` o `auto d = 42.0_mi`. No hay ninguna ventaja ni desventaja en el rendimiento de los literales definidos por el usuario; son principalmente para comodidad o para la deducción de tipos en tiempo de compilación. La biblioteca estándar tiene literales definidos por el usuario para, `std::string` `std::complex` y para las unidades de operaciones de tiempo y duración del `<chrono>` encabezado:

```
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

Para implementar un literal definido por el usuario, defina un **operador ""** en el ámbito del espacio de nombres con uno de los siguientes formatos:

```
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 a la biblioteca estándar definir 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, ya sea definido por el usuario o no, es esencialmente una secuencia de caracteres alfanuméricos, como `101`, o `54.7`, o `"hello"` o `true`. El compilador interpreta la secuencia como un entero, Float, una cadena de caracteres `const *`, etc. Un literal definido por el usuario que acepta como entrada cualquier tipo que el compilador asignado al valor literal se conoce informadamente como un *literal cocido*. 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. Puede lograr el mismo resultado de otras maneras, pero los literales definidos por el usuario son menos detallados que las alternativas.

```
// 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 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 se define acepta el literal como una secuencia de valores `char`. Depende de usted 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:

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

Los literales sin formato se pueden usar para proporcionar una interpretación personalizada de una secuencia de entrada 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 cocidos, también se pueden usar para la validación en tiempo de compilación de las secuencias de entrada.

Ejemplo: limitaciones de los 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:

```
#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;
}

```

```
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++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 estándar y de diseño estándar](#)

Vea también

[Referencia del lenguaje C++](#)

Sistema de tipos de C++

06/03/2021 • 26 minutes to read • [Edit Online](#)

El concepto de *tipo* es muy importante en C++. Cada variable, argumento de función y valor devuelto por una función debe tener un tipo para compilarse. Asimismo, antes de evaluar cada una de las expresiones (incluidos los valores literales), el compilador da implícitamente un tipo a estas expresiones. Algunos ejemplos de tipos incluyen `int` para almacenar valores enteros, `double` para almacenar valores de punto flotante (también conocidos como tipos de datos *escalares*) o la clase `STD:: basic_string` de la biblioteca estándar para almacenar texto. Puede crear su propio tipo definiendo `class` o `struct`. El tipo especifica la cantidad de memoria que se asignará para la variable (o el resultado de la expresión), las clases de valores que se pueden almacenar en esa variable, cómo se interpretan estos valores (como patrones de bits) y las operaciones que se pueden realizar en ella. Este artículo contiene información general sobre las principales características del sistema de tipos de C++.

Terminología

Variable: el nombre simbólico de una cantidad de datos para que el nombre se pueda usar para tener acceso a los datos a los que hace referencia a lo largo del ámbito del código en el que se define. En C++, la *variable* se utiliza normalmente para hacer referencia a las instancias de tipos de datos escalares, mientras que las instancias de otros tipos normalmente se denominan *objetos*.

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

Tipo Pod (datos antiguos sin formato): esta categoría informal de tipos de datos en C++ hace referencia a los tipos que son escalares (vea la sección de tipos fundamentales) o son *clases Pod*. Una clase POD no tiene ningún miembro de datos estático que no sea también POD, y no tiene ningún constructor definido por el usuario, ningún destructor definido por el usuario ni ningún operador de asignación definido 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 *fuertemente tipado* y también tiene *tipo estático*; cada objeto tiene un tipo y ese tipo nunca cambia (no debe confundirse con los objetos de datos estáticos). Al declarar una variable en el código, debe especificar explícitamente su tipo o utilizar la `auto` palabra clave para indicar al compilador que deduzca el tipo del inicializador. Al declarar una función en el código, debe especificar el tipo de cada argumento y su valor devuelto, o bien, `void` si la función no devuelve ningún valor. La excepción se produce cuando se utilizan plantillas de función, que están permitidas en los argumentos de tipos arbitrarios.

Una vez que se declara por primera vez una variable, no se puede cambiar su tipo. Sin embargo, el valor de la variable o el valor devuelto por una función se puede copiar en otra variable de distinto tipo. Estas operaciones se denominan *conversiones de tipo*, que a veces son necesarias, pero también son posibles orígenes de pérdida de datos o incorrectas.

Cuando se declara una variable de tipo POD, se recomienda encarecidamente 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. Este es un aspecto importante de C++ que debe recordarse, sobre todo si anteriormente utilizaba otro lenguaje que controlaba la inicialización sin su intervención. Cuando

se declara una variable de un tipo que pertenece una 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.

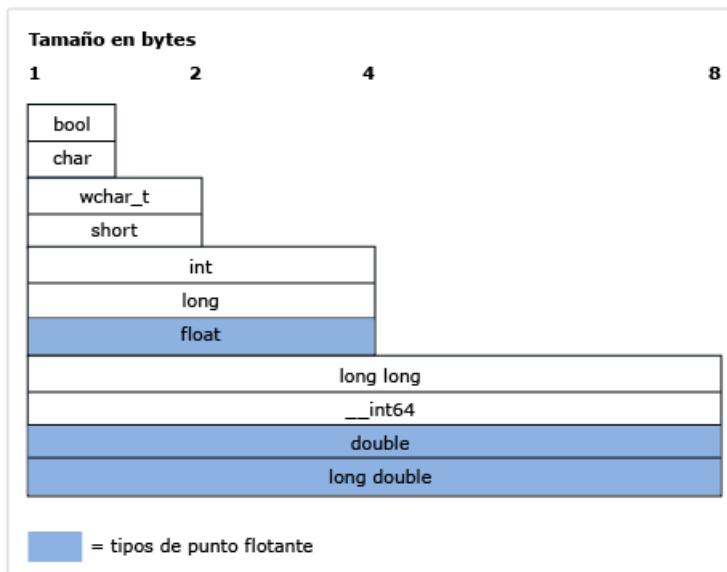
```
int result = 0;           // Declare and initialize an integer.
double coefficient = 10.8; // Declare and initialize a floating
                           // point value.
auto name = "Lady G.";   // Declare a variable and let compiler
                           // deduce the type.
auto address; 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

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 incluye muchos *tipos fundamentales*, también conocidos como *tipos integrados*. Esto incluye tipos numéricos como `int` , `double` , `long` , `bool` , además de `char` los `wchar_t` tipos y para caracteres ASCII y Unicode, respectivamente. La mayoría de los tipos fundamentales enteros (excepto `bool` , `double` , `wchar_t` y tipos relacionados) tienen `unsigned` versiones, que modifican el intervalo de valores que la variable puede almacenar. Por ejemplo, un `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 comprendido entre 0 y 4.294.967.295. El número total de valores posibles en cada caso es el mismo; solo cambia el intervalo.

El compilador reconoce los tipos fundamentales y tiene reglas integradas que rigen las operaciones que se pueden realizar en esos tipos y cómo se pueden convertir en otros tipos fundamentales. Para obtener una lista completa de los tipos integrados y sus límites de tamaño y numéricos, vea [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 enumeran los tipos fundamentales que se usan 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). Es el tipo de carácter que se utiliza en las cadenas de tipo <code>std::wstring</code> .
<code>unsigned char</code>	1 byte	C++ no tiene ningún tipo de bytes integrado. <code>unsigned char</code> Se usa para representar un valor de byte.
<code>unsigned int</code>	4 bytes	Opción predeterminada para los marcadores de bits.
<code>long long</code>	8 bytes	Representa valores enteros muy grandes.

Otras implementaciones de C++ pueden utilizar diferentes tamaños para determinados tipos numéricos. Para obtener más información sobre los tamaños y las relaciones de tamaño que requiere el estándar de C++, vea [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`), lo que a veces es necesario al asignar memoria sin formato (sin tipo). Sin embargo, los punteros a `void` no tienen seguridad de tipos y, en general, no se recomienda su uso en C++ moderno. En una declaración de función, un `void` valor devuelto significa que la función no devuelve un valor; se trata de un uso común y aceptable de `void`. Aunque el lenguaje C requiere funciones que no tienen parámetros para declarar `void` en la lista de parámetros, por ejemplo, `fou(void)` esta práctica no se recomienda en C++ moderno y debe declararse `fou()`. Para obtener más información, vea [conversiones de tipos y seguridad de tipos](#).

Calificador de tipo const

Cualquier tipo integrado o definido por el usuario se puede calificar con la palabra clave `const`. Además, las

funciones miembro pueden ser `const` calificadas e incluso `const` sobrecargadas. El valor de un `const` tipo no se puede modificar una vez inicializado.

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

El `const` calificador se usa mucho en las declaraciones de funciones y variables y la "corrección `const`" es un concepto importante en C++; básicamente significa usar `const` para garantizar, en tiempo de compilación, que los valores no se modifican de forma involuntaria. Para obtener más información, vea [const](#).

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

Tipos string

En realidad, el lenguaje C++ no tiene ningún tipo de cadena integrado; `char` y `wchar_t` almacenan caracteres individuales: debe declarar una matriz de estos tipos para aproximar una cadena, agregando un valor null final (por ejemplo, ASCII `'\0'`) al elemento de la matriz, uno 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 C++ moderno, tenemos los tipos de biblioteca estándar `std::string` (para cadenas de caracteres de tipo de 8 bits `char`) o `std::wstring` (para cadenas de caracteres de tipo de 16 bits `wchar_t`). Estos contenedores de la biblioteca estándar de C++ se pueden considerar como tipos de cadena nativas porque forman parte de las bibliotecas estándar que se incluyen en cualquier entorno de compilación de C++ compatible. Solo tiene que usar la directiva `#include <string>` 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 previamente) en C++ moderno.

Tipos definidos por el usuario

Al definir un `class`, `struct`, `union` o `enum`, esa construcción se utiliza 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. Aprende del tipo cuando encuentra por primera vez 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 obtener más información, vea [sobrecarga de funciones](#).

Tipos de puntero

Con respecto a las primeras versiones del lenguaje C, C++ sigue dejando que 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, se hace referencia a estos como *punteros sin formato* y se tiene acceso a ellos en el código mediante operadores especiales `*` (asterisco) o `->` (Dash con mayor que). Esto se denomina *desreferenciar* y el que se usa depende de si se va a desreferenciar un puntero a un valor 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 que ayudan a usar punteros sin formato si se desea, pero en C++ moderno ya no es necesario (o se recomienda) usar punteros sin formato para la propiedad del objeto, debido a la evolución del [puntero inteligente](#) (más información al final de esta sección). Todavía resulta útil y seguro utilizar punteros sin formato para inspeccionar objetos, pero si es necesario utilizarlos para la propiedad del objeto, debe hacerse con precaución y debe valorarse cuidadosamente el modo en que los objetos de su propiedad se crean y se destruyen.

Lo primero que debe saber es que, al declarar una variable de puntero sin formato, se asignará solo la memoria necesaria para almacenar una dirección de la ubicación de memoria a la que el puntero hará referencia cuando esté desreferenciado. Todavía no se ha asignado la asignación de la memoria para el propio valor de datos (también denominado *memoria auxiliar*). Es decir, al declarar una variable de puntero sin formato, se crea una variable de la dirección de memoria, no una variable real de los datos. Si se desreferencia una variable de puntero antes de tener la seguridad de que contiene una dirección válida en una memoria auxiliar, se producirá un comportamiento no definido (normalmente un error irrecuperable) en el programa. En el siguiente ejemplo se muestra este tipo de error:

```
int* pNumber;           // Declare a pointer-to-int variable.  
*pNumber = 10;          // error. Although this may compile, it is  
// a serious error. We are dereferencing an  
// uninitialized pointer variable with no  
// allocated memory to point to.
```

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:

```
int number = 10;          // Declare and initialize a local integer  
// variable for data backing store.  
int* pNumber = &number;    // Declare and initialize a local integer  
// pointer variable to a valid memory  
// address to that backing store.  
...  
*pNumber = 41;            // Dereference and store a new value in  
// the memory pointed to by  
// pNumber, the integer variable called  
// "number". Note "number" was changed, not  
// "pNumber".
```

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, la memoria auxiliar de los punteros son los tipos definidos por el usuario que se asignan dinámicamente en un área de memoria denominada *montón* (o *almacén libre*) mediante el uso de una `new` expresión de palabra clave (en la programación de estilo C, `malloc()` se usó la antigua función de biblioteca en tiempo de ejecución de C). Una vez asignadas, estas variables se denominan normalmente objetos, sobre todo si se basan en una definición de clase. La memoria que se asigna con `new` debe eliminarse mediante una `delete` instrucción correspondiente (o, si se usó la `malloc()` función para asignarla, la función en tiempo de ejecución de C `free()`).

Sin embargo, es fácil olvidarse de eliminar un objeto asignado dinámicamente, especialmente en el código complejo, lo que provoca un error de recurso denominado *pérdida de memoria*. Por esta razón, el uso de punteros sin formato no es recomendable en el lenguaje C++ actual. Casi siempre es mejor ajustar un puntero sin formato en un [puntero inteligente](#), que libera automáticamente la memoria cuando se invoca su destructor (cuando el código sale del ámbito del puntero inteligente); mediante el uso de punteros inteligentes, se elimina prácticamente una clase completa 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();`

```
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 obtener más información sobre los punteros inteligentes, vea [punteros inteligentes](#).

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

Para obtener más información sobre los punteros en general, vea [punteros](#).

Tipos de datos de Windows

En la programación clásica de Win32 para C y C++, la mayoría de las funciones utilizan definiciones de tipo y macros específicas de Windows `#define` (definidas en `windef.h`) para especificar los tipos de parámetros y valores devueltos. Estos tipos de datos de Windows son principalmente nombres especiales (alias) asignados a tipos integrados de C/C++. Para obtener una lista completa de estas definiciones de tipo y las definiciones de preprocesador, vea [tipos de datos de Windows](#). Algunas de estas definiciones de tipo, como `HRESULT` y `LCID`, son útiles y descriptivas. Otros, 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. También hay tipos de datos especiales asociados a la biblioteca de Windows Runtime, que se enumeran como [Windows Runtime tipos de datos base](#). En el lenguaje C++ actual, la regla general establece una preferencia por los tipos fundamentales de C++, a menos que el tipo de Windows comunique un significado adicional sobre cómo debe interpretarse el valor.

Más información

Para obtener más información sobre el sistema de tipos de C++, vea los temas siguientes.

[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.

Consulta también

[Aquí está otra vez C++](#)

[Referencia del lenguaje C++](#)

[Biblioteca estándar de C++](#)

Ámbito (C++)

06/03/2021 • 8 minutes to read • [Edit Online](#)

Cuando se declara un elemento de programa como una clase, una función o una variable, su nombre solo se puede "Mostrar" y usar en determinadas partes del programa. El contexto en el que un nombre es visible se denomina **ámbito**. Por ejemplo, si declara una variable `x` en una función, `x` solo es visible dentro del cuerpo de la función. Tiene **ámbito local**. Puede tener otras variables con el mismo nombre en el programa; siempre y cuando se encuentren en ámbitos distintos, no infringirán la regla de definición única y no se producirá ningún error.

En el caso de las variables automáticas no estáticas, 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 aquél que se declara fuera de cualquier clase, función o espacio de nombres. Sin embargo, en C++ incluso existen estos nombres 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 está visible en otros archivos del programa.
- **Ámbito de espacio de nombres** Un nombre que se declara dentro de un [espacio de nombres](#), fuera de cualquier definición de clase o 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 a través de archivos diferentes.
- **Ámbito local** Un nombre declarado dentro de una función o una expresión lambda, incluidos los nombres de parámetro, tiene ámbito local. A menudo se conocen como "variables locales". Solo son visibles desde su punto de declaración hasta el final de la función o el cuerpo de la expresión 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 miembros de clase se controla más a través de las `public` `private` `protected` palabras clave, y. Solo se puede tener acceso a los miembros públicos o protegidos mediante los operadores de selección de miembro (`.` o `->`) o operadores de puntero a miembro (`*` o `->*`).
- **Ámbito de instrucción** Los nombres declarados en una `for` `if` instrucción,, `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 en todo el cuerpo de la función, incluso antes de su punto de declaración. El ámbito de función permite escribir instrucciones como `goto cleanup` antes de que `cleanup` se declare la etiqueta.

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.

```

Sample::Func(char *szWhat)
{
    int i = 0;
    cout << "i = " << i << "\n";
    {
        int i = 7, j = 9;
        cout << "i = " << i << "\n"
            << "j = " << j << "\n";
    }
    cout << "i = " << i << "\n";
}

```

El bloque interno contiene los objetos de ámbito local i y j.

El bloque externo contiene el objeto de ámbito local i y el parámetro de formato szWhat.

Ocultar el ámbito de bloque y el nombre

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

```

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

```

NOTE

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, todavía se puede tener acceso al nombre de clase cuando el prefijo es la palabra clave `class`.

```

// 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

```

NOTE

Cualquier lugar en el que se llame al nombre de clase (`Account`), la clase de palabra clave debe usarse para diferenciarlo de la cuenta de variable de 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 `class` palabra clave:

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

El del `Account` inicializador (entre paréntesis) de la instrucción anterior tiene ámbito global, es de tipo `double` .

NOTE

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 objetos de clase, vea [clases, estructuras y uniones](#). Para obtener información sobre el uso de los `new` `delete` operadores y de almacenamiento libre, vea [operadores New y DELETE](#).

Ocultar nombres con ámbito global

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

```
#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";
}
```

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

Consulta también

[Conceptos básicos](#)

Archivos de encabezado (C++)

06/03/2021 • 8 minutes to read • [Edit Online](#)

Los nombres de los elementos de programa, como variables, funciones, clases, etc., se deben declarar antes de que se puedan utilizar. Por ejemplo, no puede simplemente escribir `x = 42` sin declarar primero "x".

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

La declaración indica al compilador si el elemento es una `int`, una `double`, una **función**, una `class` u otra cosa. Además, cada nombre se debe declarar (directa o indirectamente) en cada archivo. cpp en el que se usa. 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 una función o una 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 producirá errores, o un comportamiento no deseado, cuando el vinculador intente fusionar mediante combinación todas las unidades de compilación en un único programa.

Para minimizar la posibilidad de errores, C++ ha adoptado la Convención de usar *archivos de encabezado* para contener declaraciones. Las declaraciones se realizan en un archivo de encabezado y, a continuación, se usa la Directiva `#include` en todos los archivos. cpp u otro archivo de encabezado que requiera dicha declaración. La Directiva `#include` inserta una copia del archivo de encabezado directamente en el archivo. cpp antes de la compilación.

NOTE

En Visual Studio 2019, la característica de *módulos* de c++ 20 se presenta como una mejora y un reemplazo eventual de los archivos de encabezado. Para obtener más información, vea [información general sobre los módulos de C++](#).

Ejemplo

En el ejemplo siguiente se 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. `do_something` no se ha definido la función miembro:

```
// 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). Llamaremos al archivo `my_class.cpp` y proporcionaremos una definición para la declaración de miembro.

Agregamos una `#include` Directiva para el archivo "my_class.h" para que la declaración de my_class se inserte en este punto en el archivo. cpp, y se incluye para realizar la `<iostream>` extracción en 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 código fuente, y se usan corchetes angulares para los encabezados de la biblioteca estándar. Además, muchos encabezados de la biblioteca estándar no tienen. h ni cualquier otra extensión de archivo.

En el archivo de implementación, se puede usar opcionalmente una `using` instrucción para evitar tener que calificar cada mención de "my_class" o "cout" con "N::" o "STD::". No coloque `using` instrucciones en los archivos de encabezado.

```
// 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. #Include 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()`.

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

using namespace N;

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

Una vez que el compilador termina de compilar cada archivo. cpp en archivos. obj, pasa los archivos. obj al enlazador. Cuando el vinculador combina los archivos 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 protecciones

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

```
// 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 se debe incluir en un archivo de encabezado

Dado que un archivo de encabezado podría estar incluido potencialmente en varios archivos, no puede contener definiciones que puedan generar varias definiciones del mismo nombre. Los siguientes elementos no están permitidos o se consideran muy incorrectos:

- definiciones de tipos integrados en un espacio de nombres o ámbito global
- definiciones de funciones no insertadas
- definiciones de variables no const
- definiciones de agregado
- espacios de nombres sin nombre
- Directivas using

El uso de la `using` Directiva no producirá necesariamente un error, pero puede causar un problema, ya que pone el espacio de nombres en el ámbito de cada archivo. cpp que incluye 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:

```

// 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

06/03/2021 • 6 minutes to read • [Edit Online](#)

En un programa de C++, un *símbolo*, por ejemplo un nombre de variable o función, se puede declarar cualquier número de veces dentro de su ámbito, pero solo se puede definir una vez. Esta regla es la "regla de definición única" (ODR). Una *declaración* introduce (o vuelve a introducir) un nombre en el programa. Una *definición* introduce un nombre. Si el nombre representa una variable, una definición la inicializa explícitamente. Una *definición de función* se compone 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 clase. (Los cuerpos de las funciones miembro se pueden definir opcionalmente de forma independiente en otro archivo).

En el ejemplo siguiente se muestran algunas declaraciones:

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

En el ejemplo siguiente se muestran algunas definiciones:

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

Un programa consta de una o varias *unidades de traducción*. Una unidad de traducción se compone 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*. Los archivos de encabezado suelen tener una extensión de *h* o *HPP*. El compilador compila cada unidad de traducción de forma independiente. Una vez completada la compilación, el vinculador combina las unidades de traducción compiladas en un único *programa*. Las infracciones de la regla ODR suelen aparecer como errores del vinculador. Los errores del enlazador se producen cuando el mismo nombre tiene dos definiciones diferentes en unidades de traducción diferentes.

En general, la mejor manera de hacer que una variable sea visible en varios archivos es colocarla en un archivo de encabezado. A continuación, agregue una directiva de `#include` en cada archivo *CPP* que requiera la declaración. Al agregar las *protecciones include* en torno al contenido del encabezado, se asegura de que los nombres que declara se definen solo una vez.

En C++ 20, los *módulos* se presentan como una alternativa mejorada a los archivos de encabezado.

En algunos casos, puede ser necesario declarar una variable global o una clase en un archivo *CPP*. En esos casos, necesita una manera de indicar al compilador y vinculador qué tipo de *vinculación* tiene el nombre. El tipo de vinculación especifica si el nombre del objeto se aplica solo al archivo o a 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 de inclusión, como en las definiciones de función o de clase.

Vinculación externa frente a interna

Una *función gratuita* es una función que se define en un ámbito global o de espacio de nombres. De forma predeterminada, las variables globales no const y las funciones libres tienen *vinculación externa*. Están visibles desde cualquier unidad de traducción del programa. Por lo tanto, ningún otro objeto global puede tener ese nombre. Un símbolo con *vinculación interna* o *sin vinculación* solo es visible dentro de la unidad de traducción en la que se declara. Cuando un nombre tiene vinculación interna, puede existir el mismo nombre 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 si lo declara explícitamente como `static`. Esto limita su visibilidad a la misma unidad de traducción en la que se declara. En este contexto, `static` significa algo distinto de cuando se aplica a variables locales.

Los objetos siguientes tienen vinculación interna de forma predeterminada:

- `const` (objetos)
- objetos `constexpr`
- `typedefs`
- objetos estáticos en el ámbito de espacio de nombres

Para proporcionar una vinculación externa a un objeto `const`, déclárela como `extern` y asígnale un valor:

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

Consulte [extern](#) para obtener más información.

Consulta también

[Conceptos básicos](#)

main argumentos de la función y de la línea de comandos

06/03/2021 • 16 minutes to read • [Edit Online](#)

Todos los programas de C++ deben tener una `main` función. Si intenta compilar un programa de C++ sin una `main` función, el compilador genera un error. (Las bibliotecas de vínculos dinámicos y las static bibliotecas no tienen una `main` función). La `main` función es donde comienza la ejecución del código fuente, pero antes de que un programa entre `main` en la función, todos los static miembros de clase sin inicializadores explícitos se establecen en cero. En Microsoft C++, los static objetos globales también se inicializan antes de la entrada a `main`. Se aplican varias restricciones a la `main` función que no se aplica 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`.
- No se puede tomar su dirección.
- No se puede llamar desde el programa.

Firma de la `main` función

La `main` función no tiene una declaración, porque está integrada en el lenguaje. En caso de que lo hiciera, la sintaxis de declaración de `main` sería similar a la siguiente:

```
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 de `main` permiten un práctico análisis de la línea de comandos de los argumentos. El lenguaje define los tipos `argc` y `argv`. Los nombres `argc` y `argv` son tradicionales, pero puede asignarles el nombre que desee.

Las definiciones de los argumentos son las siguientes:

`argc`

Entero que contiene el recuento de argumentos que siguen en `argv`. El `argc` parámetro siempre es 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 línea de comandos, vea [personalizar el procesamiento de línea de comandos de C++](#).

NOTE

Por Convención, `argv[0]` es el nombre de archivo del programa. Sin embargo, en Windows es posible generar un proceso mediante el uso de `CreateProcess`. Si usa los argumentos primero y segundo (`LpApplicationName` y `LpCommandLine`), `argv[0]` es posible que no sea 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 secciones siguientes se describe el comportamiento específico de Microsoft.

La `wmain` función y la `_tmain` macro

Si diseña el código fuente para usar Unicode Wide char acters, puede usar el punto de entrada específico de Microsoft `wmain`, que es la versión Wide- char acter de `main`. Esta es la sintaxis de declaración efectiva para `wmain`:

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

También puede usar el específico de Microsoft `_tmain`, que es una macro de preprocesador definida en `tchar.h`. `_tmain` se resuelve como a `main` menos que `_UNICODE` se defina. En ese caso, `_tmain` se resuelve en `wmain`. La `_tmain` macro y otras macros que comienzan por `_t` son útiles para el código que debe compilar versiones independientes para conjuntos acter estrechos y anchos char . Para obtener más información, vea [usar asignaciones de texto genérico](#).

Devolver `void` desde main

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

Si declara `main` o `wmain` como devuelve `void`, no puede devolver un exit código al proceso primario o al sistema operativo mediante el uso de una `return` instrucción. Para devolver un exit código cuando `main` o `wmain` se declara como `void`, debe usar la `exit` función.

`envp` Argumento de la línea de comandos

Las `main` `wmain` firmas o permiten una extensión opcional específica de Microsoft para el acceso a las variables de entorno. Esta extensión también es común en otros compiladores para sistemas Windows y UNIX. El nombre `envp` es tradicional, pero puede asignar el nombre que desee al parámetro de entorno. Estas son las declaraciones eficaces de las listas de argumentos que incluyen el parámetro Environment:

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

`envp`

El `envp` parámetro opcional es una matriz de cadenas que representan las variables establecidas en el entorno del 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 el programa utiliza `wmain` en lugar de `main`, utilice el `wchar_t` tipo de datos en lugar de `char`.

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

Argumentos de ejemplo para `main`

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

```
// 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 utilizadas por el código de Microsoft C/C++ son específicas de Microsoft. El código de inicio en tiempo 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]`. Los elementos rodeados por comillas dobles impiden la interpretación de un espacio o una tabulación como el 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 espacios en blanco. Se puede incrustar una cadena entre comillas dentro de un argumento. El símbolo de intercalación (`\`) no se reconoce como un char acter o delimitador de escape. 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, todas las char acters leídas 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 marca de comillas dobles se interpreta como una secuencia de escape mediante la main barra diagonal inversa, lo que hace que se coloque una marca de comilla doble literal (") argv .

Ejemplo de análisis de argumentos de línea de comandos

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

```
// 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 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]	ARGV3
"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 le permite usar actores comodín char , el signo de interrogación (?) y el asterisco (*), para especificar los argumentos de nombre de archivo y ruta de acceso en la línea de comandos.

Los argumentos de línea de comandos se controlan mediante una rutina interna en el código de inicio en tiempo de ejecución, que, de forma predeterminada, no expande los caracteres comodín en cadenas independientes en la `argv` matriz de cadenas. Puede habilitar la expansión de caracteres comodín incluyendo el `setargv.obj` archivo (`wsetargv.obj` archivo para `wmain`) en las `/link` Opciones del compilador o en la línea de `LINK` comandos.

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

Personalizar el 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 de C++

06/03/2021 • 5 minutes to read • [Edit Online](#)

En C++, puede salir de un programa de las siguientes maneras:

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

Función `exit`

La `exit` función, declarada en `<stdlib.h>`, finaliza un programa de C++. El valor proporcionado como argumento para `exit` se devuelve al sistema operativo como el código de retorno o el código de salida del programa. Por convención, un código de retorno de cero significa que el programa se completó correctamente. Puede usar las constantes `EXIT_FAILURE` y `EXIT_SUCCESS`, también definidas en `<stdlib.h>`, para indicar si el programa se ha realizado correctamente o no.

Emitir una `return` instrucción desde la `main` función es equivalente a llamar a la `exit` función con el valor devuelto como su argumento.

Función `abort`

La `abort` función, también declarada 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 llama a los destructores de objeto globales), pero `abort` finaliza el programa inmediatamente. La `abort` función omite el proceso de destrucción normal de los objetos estáticos globales inicializados. También omite cualquier procesamiento especial que se especificó mediante la `atexit` función.

Función `atexit`

Utilice la `atexit` función para especificar las acciones que se ejecutan antes de que finalice el programa. Ninguno de los objetos estáticos globales inicializados antes de la llamada a `atexit` se destruye antes de la ejecución de la función de procesamiento de salida.

`return` instrucción en `main`

Emitir una `return` instrucción desde `main` es funcionalmente equivalente a llamar a la `exit` función.

Considere el ejemplo siguiente:

```
// return_statement.cpp
#include <stdlib.h>
int main()
{
    exit( 3 );
    return 3;
}
```

Las `exit` instrucciones y del `return` ejemplo anterior son funcionalmente idénticas. Normalmente, C++

requiere que las funciones que tienen tipos de valor devuelto distintos de `void` devuelvan un valor. La `main` función es una excepción; puede finalizar sin una `return` instrucción. En ese caso, devuelve un valor específico de la implementación para el proceso de invocación. La `return` instrucción le permite especificar un valor devuelto de `main`.

Destrucción de objetos estáticos

Cuando se llama a `exit` o se ejecuta una `return` instrucción de `main`, los objetos estáticos se destruyen en el orden inverso a su inicialización (después de la llamada a `atexit` si existe). 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 este programa termine de usar la `return` instrucción, primero `sd2` se destruye y despues `sd1`. El destructor para la clase de `ShowData` cierra los archivos asociados a estos objetos estáticos.

```
// 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:

```
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

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

Lvalues y rvalues (C++)

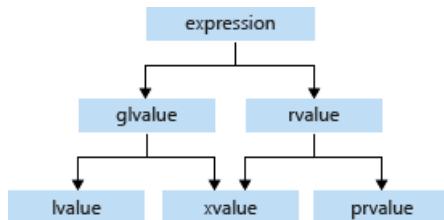
06/03/2021 • 4 minutes to read • [Edit Online](#)

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 deben seguir los compiladores al crear, copiar y mover objetos temporales durante la evaluación de expresiones.

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

- Un *glvalue* es una expresión cuya evaluación determina la identidad de un objeto, un campo de bits o una función.
- Un *prvalue* es una expresión cuya evaluación Inicializa un objeto o un campo de bits, o calcula el valor del operando de un operador, según se especifica en el contexto en el que aparece.
- Un *xValue* es un glvalue que denota un objeto o un campo de bits cuyos recursos se pueden reutilizar (normalmente porque está cerca del final de su duración). Ejemplo: ciertos tipos de expresiones que afectan a las referencias rvalue (8.3.2) producen xvalues, 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 *l* es un glvalue que no es un xValue.
- Un valor *r* es un prvalue o un xValue.

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



Un valor *l* tiene una dirección a la que el programa puede tener acceso. Entre los ejemplos de expresiones lvalue se incluyen nombres de variable, como `const` variables, elementos de matriz, llamadas a funciones que devuelven una referencia lvalue, campos de bits, uniones y miembros de clase.

Una expresión prvalue no tiene ninguna dirección a la que pueda acceder el programa. Entre los ejemplos de expresiones prvalue se incluyen los literales, las llamadas de función que devuelven un tipo sin referencia y los objetos temporales que se crean durante la evaluación de la expresión, pero solo el compilador puede acceder a ellos.

Una expresión xValue tiene una dirección que ya no es accesible para el programa, pero se puede usar para inicializar una referencia rvalue, que proporciona acceso a la expresión. Entre los ejemplos se incluyen las llamadas a funciones que devuelven una referencia rvalue y las expresiones subscript, miembro y puntero a miembro de matriz, donde 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:

```
// 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
}
```

NOTE

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* se suelen usar cuando se hace referencia a referencias a objetos. Para obtener más información sobre las referencias, vea [declarador de referencia de valor L: &](#) y [declarador de referencia de valor r: &&](#).

Consulta también

[Conceptos básicos](#)

[Declarador de referencia a un valor L: &](#)

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

Objetos temporales

06/03/2021 • 4 minutes to read • [Edit Online](#)

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

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

```
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 *temporal* `+ Complex3` y el resultado se copia en `ComplexResult` (siempre 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. Los puntos en los que se produce la destrucción se muestran en la tabla siguiente.

Puntos de destrucción de objetos temporales

MOTIVO POR EL QUE SE CREÓ EL OBJETO TEMPORAL	PUNTO DE DESTRUCCIÓN
Resultado de evaluación de la expresión	Todos los objetos temporales creados como resultado de la evaluación de expresiones se destruyen al final de la instrucción de expresión (es decir, en el punto y coma) o al final de las expresiones de control de las <code>for</code> <code>if</code> instrucciones,, <code>while</code> <code>do</code> y <code>switch</code> .

MOTIVO POR EL QUE SE CREÓ EL OBJETO TEMPORAL	PUNTO DE DESTRUCCIÓN
Inicializar <code>const</code> referencias	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 y se inicializa con la expresión de inicialización. Este objeto temporal se destruye inmediatamente después de que se destruya el objeto de referencia al que está enlazado.

Alignment

02/11/2020 • 8 minutes to read • [Edit Online](#)

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 struct 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, nunca tendrá que preocuparse de la alineación, ya que la alineación predeterminada ya es óptima. En algunos casos, sin embargo, puede lograr mejoras significativas en el rendimiento o ahorrar memoria si especifica una alineación personalizada para las estructuras de datos. Antes de Visual Studio 2015 podría usar las palabras clave específicas de Microsoft `_alignof` y `_declspec(align)` para especificar una alineación mayor que la predeterminada. A partir de Visual Studio 2015, debe usar las palabras clave estándar de C++ 11 `alignof` y `alignas` para obtener la máxima portabilidad del código. Las nuevas palabras clave se comportan de la misma manera bajo el capó que las extensiones específicas de Microsoft. La documentación de esas extensiones también se aplica a las nuevas palabras clave. Para obtener más información, vea [alignof operador](#) y [align](#). El estándar de C++ no especifica el comportamiento del empaquetado para la alineación en los límites menores que el valor predeterminado del compilador para la plataforma de destino, por lo que todavía necesitará usar Microsoft `#pragma pack` en ese caso.

Utilice la [clase aligned_storage](#) para la asignación de memoria de estructuras de datos con alineaciones personalizadas. La [clase aligned_union](#) es para especificar la alineación de las 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. Se dice que esa dirección está alineada con $4N + 3$, donde 4 indica la potencia elegida de 2. La alineación de una dirección depende de la potencia elegida de 2. 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 operan en los datos almacenados en memoria. Los datos se identifican por sus direcciones en la memoria. Un único dato también tiene un tamaño. Se llama a un dato de referencia *Naturally aligned* si su dirección está alineada con su tamaño. En caso contrario, se llama *desalineado*. Por ejemplo, una referencia de punto flotante de 8 bytes se alinea de forma natural si la dirección usada para identificarla tiene una alineación de 8 bytes.

Control del compilador de la alineación de datos

Los compiladores intentan realizar asignaciones de datos de forma que se evite la alineación incorrecta de los 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 rellena las estructuras de una manera que alinea naturalmente cada elemento de la estructura. Considere la estructura `struct x` en el ejemplo de código siguiente:

```

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

```

// 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 un valor `sizeof(struct x_)` de 12 bytes.

La segunda declaración incluye dos elementos de relleno:

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

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

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

adr	offset	element
0x0000		---
0x0000	char a;	// bar[0]
0x0001	char pad0[3];	
0x0004	int b;	
0x0008	short c;	
0x000a	char d;	
0x000b	char _pad1[1];	
0x000c		---
0x000c	char a;	// bar[1]
0x000d	char _pad0[3];	
0x0010	int b;	
0x0014	short c;	
0x0016	char d;	
0x0017	char _pad1[1];	
0x0018		---
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 es una manera portátil estándar de C++ de especificar la alineación personalizada de variables y tipos definidos por el usuario. El operador es igualmente una forma estándar y portátil de obtener la alineación de un tipo o variable especificado.`

Ejemplo

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

```
// 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 la estructura de datos](#)

Tipos triviales, de diseño estándar, POD y literales

06/03/2021 • 10 minutes to read • [Edit Online](#)

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.

```

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:

```

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:

```

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:

```

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:

```

#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.

Consulta también

[Conceptos básicos](#)

Clases C++ como tipos de valor

06/03/2021 • 6 minutes to read • [Edit Online](#)

Las clases de C++ son tipos de valor predeterminados. Se pueden especificar como tipos de referencia, lo que permite que el comportamiento polimórfico admita la programación orientada a objetos. A veces, los tipos de valor se ven desde la perspectiva de la memoria y el control de diseño, mientras que los tipos de referencia son acerca de las clases base y las funciones virtuales para propósitos 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. En el caso de los tipos de referencia, hace que la clase sea no copiable (deshabilite el constructor de copias y el operador de asignación de copia) y use un destructor virtual, que admite el polimorfismo previsto. Los tipos de valor también son sobre el contenido, que, cuando se copian, siempre proporcionan dos valores independientes que se pueden modificar por separado. Tipos de referencia sobre identidad: ¿Qué tipo de objeto es? Por este motivo, los "tipos de referencia" también se conocen como "tipos polimórficos".

Si realmente desea un tipo de referencia (clase base, funciones virtuales), debe deshabilitar explícitamente la copia, como se muestra en la `MyRefType` clase en el código siguiente.

```
// cl /EHsc /nologo /W4

class MyRefType {
private:
    MyRefType & operator=(const MyRefType &);
    MyRefType(const MyRefType &);

public:
    MyRefType () {}
};

int main()
{
    MyRefType Data1, Data2;
    // ...
    Data1 = Data2;
}
```

Al compilar el código anterior se producirá el siguiente error:

```
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

La sobrecarga de asignación de copia se evita debido a nuevas optimizaciones de copia. Por ejemplo, al insertar una cadena en medio de un vector de cadenas, no habrá ninguna sobrecarga de reasignación de copia, solo un movimiento, incluso si se produce un aumento del propio vector. Esto también se aplica a otras operaciones, por ejemplo, realizar una operación de agregar en dos objetos de gran tamaño. ¿Cómo se habilitan estas optimizaciones de operaciones de valores? En algunos compiladores de C++, el compilador lo habilitará de manera implícita, al igual que el compilador puede generar automáticamente los constructores de copia. Sin embargo, en C++, la clase deberá "participar" para trasladar la asignación y los constructores declarándolos en la definición de clase. Esto se logra mediante el uso de la referencia rvalue de signo de y comercial (&&) en las

declaraciones de función miembro apropiadas y la definición de los métodos de asignación de movimiento y del constructor de movimiento. También debe insertar el código correcto para "robar el Trip" del objeto de origen.

¿Cómo se decide si se necesita el traslado habilitado? Si ya sabe que necesita la construcción de copias habilitada, es probable que desee habilitar el movimiento si puede ser más barato que una copia en profundidad. Sin embargo, si sabe que necesita la compatibilidad con Move, no significa necesariamente que desee que la copia esté habilitada. Este último caso se denominaría "tipo de solo movimiento". Un ejemplo que ya está en la biblioteca estándar es `unique_ptr`. Como nota lateral, el antiguo `auto_ptr` está en desuso y se ha reemplazado con `unique_ptr` precisión debido a la falta de compatibilidad con la semántica de transferencia en la versión anterior de C++.

Mediante el uso de la semántica de movimiento, puede devolver por valor o insertar en el medio. Move es una optimización de la copia. La asignación del montón es necesaria como solución alternativa. Considere el siguiente seudocódigo:

```
#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
```

Habilitar Move para tipos de valor adecuados

En el caso de una clase de valor, donde el movimiento puede ser más barato que una copia en profundidad, habilite la construcción de movimiento y la asignación de movimiento para mayor eficacia. Considere el siguiente seudocódigo:

```
#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/asignación de copia, habilite también la construcción o asignación de movimiento si puede ser más barata 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`.

Consulta 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

16/04/2021 • 17 minutes to read • [Edit Online](#)

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, es posible que tenga que almacenar el resultado de una operación de punto flotante en una variable de tipo o que tenga que pasar el valor de a una función que `int` `unsigned int` toma `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 conversiones estándar *integradas* para convertir uno de los operandos para que coincidan los tipos. 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 compila el programa. Para obtener más información sobre las reglas que rigen las conversiones estándar, vea [Conversiones estándar](#). Para obtener más información sobre las conversiones definidas por el usuario, vea [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 en modo silencioso y no emite advertencias. Las conversiones siguientes son de ampliación.

FROM	EN
Cualquier <code>signed</code> tipo entero o excepto <code>unsigned</code> <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>

FROM	EN
<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 es segura, agregue al código algún tipo de comprobación en tiempo de ejecución para controlar la posible pérdida de datos para que no haga que el programa produzca resultados incorrectos.

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.

```
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 = 0xffff; //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.

```
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 cuando se almacena en `unsigned int` `num`. Esto hace que el patrón de bits se reinterprete.

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

El compilador no advierte sobre las conversiones implícitas entre tipos enteros con signo y sin signo. Por lo tanto, se recomienda evitar por completo las conversiones de firma a sin signo. Si no puede evitarlos, agregue una comprobación en tiempo de ejecución para detectar si el valor que se va a convertir 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 siguiente ejemplo de código mal diseñado parece sin sentido y, sin embargo, compilará y producirá un resultado de "p". En primer lugar, el literal constante de cadena "Help" se convierte en un que apunta al primer elemento de la matriz; ese puntero se incrementa en tres elementos para que ahora apunta al último elemento `char* 'p'`.

```
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 producirá un error en algunos casos si los dos tipos no están completamente relacionados, pero en otros casos no producirá un error aunque la operación no sea segura para 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. Un uso eficaz de una conversión es cuando el código realiza una conversión de limitación y sabe que la conversión no hace que el programa produzca resultados incorrectos. 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 convertir la constidad de una variable para pasársela a una función que requiere un argumento `no 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.

```
(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 son malo porque son difíciles de reconocer de un vistazo o de buscar, y son lo suficientemente 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 está intentando convertir entre tipos completamente 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.

```

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 más información, consulte [static_cast](#).

- `dynamic_cast`, para las conversión seguras y comprobadas en tiempo de ejecución de puntero a base a puntero a derivado. Un es más seguro que un para las bajadas, pero la comprobación en tiempo de `dynamic_cast` ejecución `static_cast` incurre en cierta sobrecarga.

```

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 más información, consulte [dynamic_cast](#).

- `const_cast`, para convertir `const` el valor -ness de una variable o convertir una variable que no `const` sea de `. const`. La conversión de -ness mediante este operador es tan propensa a errores como el uso de una conversión de estilo C, salvo que con usted es menos probable que realice la conversión `const` `const_cast` accidentalmente. A veces, tiene que convertir el valor -ness de una variable, por ejemplo, para pasar una variable a una función que toma un parámetro que no `const` `const` es de `const`. El ejemplo siguiente muestra cómo hacerlo.

```

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

```

Para más información, consulte [const_cast](#).

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

NOTE

Este operador de conversión no se usa con tanta frecuencia como los demás y no se garantiza que sea portátil para otros compiladores.

En el ejemplo siguiente se muestra cómo `reinterpret_cast` difiere de `static_cast`.

```
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 obtener más información, vea [reinterpret_cast Operador](#).

Consulte también

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

[Aquí está otra vez C++](#)

[Referencia del lenguaje C++](#)

[Biblioteca estándar de C++](#)

Conversiones estándar

06/03/2021 • 25 minutes to read • [Edit Online](#)

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

NOTE

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):

```
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_num1;
```

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. Sin embargo, 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 en otro tipo entero más amplio, es decir, un tipo que puede representar un conjunto mayor de valores. Este tipo de ampliación de conversión se denomina *promoción de entero*. Con la promoción de entero, puede usar los siguientes tipos en una expresión siempre que se pueda usar otro tipo entero:

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

- `int` campos de bits
- Enumerators

Las promociones de C++ son "que conserven los valores", ya que se garantiza que el valor después de la promoción sea el mismo que el valor antes de la promoción. En las promociones de preservación de valores, los objetos de tipos enteros más cortos (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 utilizada por el estándar C, las conversiones de preservación de valores no conservan el "signo" 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. Sin embargo, pueden generar resultados diferentes si el objeto promocionado aparece como:

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

Estos operadores dependen del signo para determinar el resultado. Las promociones de conservación de valores y de conservación de la firma producen resultados diferentes cuando se aplican a estos operandos.

- Operando izquierdo de `>>` o `>>=`

Estos operadores tratan las cantidades con signo y sin signo de manera diferente en una operación de desplazamiento. En el caso de las cantidades con signo, una operación de desplazamiento a la derecha propaga el bit de signo en las posiciones de bits vacantes, mientras que las posiciones de bits vacantes se rellenan con ceros en cantidades sin signo.

- Un argumento para una función sobrecargada, o el operando de un operador sobrecargado, que depende de la firma del tipo de operando para la coincidencia de argumentos. Para obtener más información sobre cómo definir operadores sobrecargados, vea [operadores sobrecargados](#).

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` se pueden usar como una abreviatura de `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 producen estas conversiones, el patrón de bits real no cambia. Sin embargo, la interpretación de los datos cambia. Considere este código:

```
#include <iostream>

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

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

En el ejemplo anterior, `signed short` `i` se define y se inicializa en un número negativo. La expresión `(u = i)` hace que `i` se convierta en un `unsigned short` objeto 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. Sin embargo, si el valor sin signo está fuera del intervalo representable del tipo con signo, el resultado no tendrá el valor correcto, tal y como se muestra en el ejemplo siguiente:

```
#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 `unsigned short` objeto entero que se debe convertir en una cantidad firmada para evaluar la expresión `(i = u)`. Dado que su valor no se puede representar correctamente en una `signed short`, los datos se interpretan erróneamente 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 un tipo flotante también se puede convertir en un tipo menos preciso, si se encuentra en un intervalo representable por ese tipo. (Vea [límites flotantes](#) para los intervalos de tipos flotantes). Si el valor original no se puede representar exactamente, se puede convertir al siguiente valor más alto o al siguiente inferior. El resultado es indefinido si no existe tal valor. Considere el ejemplo siguiente:

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

El valor máximo que se represente por tipo `float` es 3,402823466E38, un número mucho menor que 1E300. Por lo tanto, el número se convierte en 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 en un tipo flotante y el valor original no se puede representar exactamente, el resultado es el siguiente valor más alto o el siguiente valor inferior que se puede representar.

Cuando un objeto de tipo flotante se convierte en un tipo entero, se *trunca* la parte fraccionaria 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 más bajo que se va a representar, el resultado es indefinido.

Conversiones aritméticas

Muchos operadores binarios (descritos en [expresiones con operadores binarios](#)) producen conversiones de operandos y producen resultados de la misma manera. La conversión de estos operadores se denomina *conversiones aritméticas habituales*. 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> .	Otro operando se convierte al tipo <code>long double</code> .
La condición anterior no se cumple y cualquiera de los operandos es de tipo <code>double</code> .	Otro operando se convierte al tipo <code>double</code> .
No se cumplen las condiciones anteriores y cualquiera de los operandos es de tipo <code>float</code> .	Otro operando se convierte al tipo <code>float</code> .
Las condiciones anteriores no se satisfacen (ninguno de los operandos es de tipo flotante).	<p>Los operandos obtienen promociones enteras 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 no se cumplen las dos condiciones anteriores, y si alguno de los operandos es de tipo <code>long</code>, el otro operando se convierte al tipo <code>long</code>. -Si no se cumplen las tres condiciones anteriores, y si 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:

```

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`. Por lo tanto, el otro operando, `iVal`, se convierte al tipo `unsigned int`. A continuación, el resultado se asigna a `dVal`. La condición que se cumple aquí es que un operando es de tipo `double`, por lo que el `unsigned int` resultado de la multiplicación se convierte al tipo `double`.

La segunda instrucción del ejemplo anterior muestra la adición de un `float` y un tipo entero: `fVal` y `ulVal`. La `ulVal` variable se convierte al tipo `float` (tercera condición de la tabla). El resultado de la suma se convierte al tipo `double` (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 ambiguas, vea [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.

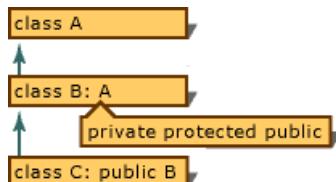


Gráfico de herencia para ilustrar la accesibilidad de clase base

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 UNA * LEGAL?
Función externa (no de ámbito de clase)	Privados	No
	Protegido	No
	Público	Sí
Función miembro B (en ámbito B)	Privados	Sí
	Protegido	Sí
	Público	Sí
Función miembro C (en ámbito C)	Privados	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 tipos explícitas, vea [operador de conversión explícita de tipos](#).

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

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

```

// 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 tanto, son inaccesibles a través de `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 se puede convertir al tipo `void *`, si el tipo `void *` es lo suficientemente grande como para contener ese puntero.

De puntero a void

Los punteros al tipo `void` se pueden convertir en 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 en un puntero a `void` (implícitamente) 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 es `const` o se `volatile` puede convertir implícitamente en un puntero de tipo `void *`.

punteros `const` y `volatile`

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

NOTE

Los punteros de C++ a los miembros, 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 tal conversión de expresión a un tipo de puntero, se convierte en un puntero denominado *puntero nulo*. Este puntero siempre compara distinto a un puntero a cualquier objeto o función válidos. Una excepción son los punteros a objetos basados en, que pueden tener el mismo desplazamiento y seguir señalando a objetos diferentes.

En C++ 11, el tipo `nullptr` debe ser preferible 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:

```
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 utiliza 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:

- La clase base especificada es accesible.
- La conversión no es ambigua. (Para obtener más información sobre las referencias de clase base ambiguas, vea [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 evalúa 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.

punteros NULL a las conversiones de miembros

Una expresión constante entera que se evalúa como cero se convierte en un puntero nulo. Este puntero siempre compara distinto a un puntero a cualquier objeto o función válidos. Una excepción son los punteros a objetos basados en, que pueden tener el mismo desplazamiento y seguir señalando 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.

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

int A::*pai = 0;

int main()
{}
```

Consulta también

[Referencia del lenguaje C++](#)

Tipos integrados (C++)

06/03/2021 • 11 minutes to read • [Edit Online](#)

Los tipos integrados (también denominados *tipos fundamentales*) se especifican mediante el estándar del lenguaje C++ y se integran 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 enteros representan números enteros. Los tipos de punto flotante pueden especificar valores que pueden tener partes fraccionarias. La mayoría de los tipos integrados se tratan como tipos distintos en el compilador. Sin embargo, algunos tipos son *sinónimos* o se tratan como tipos equivalentes por el compilador.

Un tipo void

El `void` tipo describe un conjunto de valores vacío. No se puede especificar ninguna variable de tipo `void`. El `void` tipo se utiliza principalmente para declarar funciones que no devuelven valores o para declarar punteros genéricos a datos sin tipo o con 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, vea [expresiones](#)).
- El operando izquierdo del operador de coma. (Para obtener más información, vea [operador de coma](#)).
- El segundo o tercer operando del operador condicional (`? :`). (Para obtener más información, vea [expresiones con el operador condicional](#)).

STD:: nullptr_t

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

Tipo booleano

El `bool` tipo puede tener valores `true` y `false`. El tamaño del `bool` tipo es específico de la implementación. Vea [tamaños de tipos integrados](#) para obtener información detallada sobre la implementación específica de Microsoft.

Tipos de caracteres

El `char` tipo es un tipo de representación de caracteres que codifica de forma eficaz los miembros del juego de caracteres de ejecución básico. El compilador de C++ trata las variables de 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 de tipo de `signed char` forma predeterminada, a menos que `/J` se use la opción de compilación. En este caso, se tratan como tipo `unsigned char` y se promueven a `int` sin extensión de signo.

Una variable de tipo `wchar_t` es un carácter ancho o multibyte. Utilice el `L` prefijo 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 puede usar `/Zc:wchar_t-` para crear `wchar_t` una definición de tipos para `unsigned short`. El `_wchar_t` tipo es un sinónimo específico de Microsoft para el `wchar_t` tipo nativo.

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

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

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

Tipos de punto flotante

Los tipos de punto flotante usan una representación IEEE-754 para proporcionar una aproximación de los valores fraccionarios en una amplia gama de magnitudes. En la tabla siguiente se enumeran los tipos de punto flotante en C++ y las restricciones comparativas en los tamaños de tipo de punto flotante. Estas restricciones las asigna 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>	Type <code>float</code> es el tipo de punto flotante más pequeño en C++.
<code>double</code>	<code>double</code> El tipo es un tipo de punto flotante que es mayor o igual que <code>float</code> el tipo, pero menor o igual que el tamaño del tipo <code>long double</code> .
<code>long double</code>	<code>long double</code> El tipo es un tipo de punto flotante que es mayor o igual que el tipo <code>double</code> .

Específico de Microsoft: la representación de `long double` y `double` es idéntica. Sin embargo, `long double` y `double` se tratan como tipos distintos en el compilador. El compilador de Microsoft C++ usa las representaciones de punto flotante de 4 y 8 754 bytes. Para obtener más información, consulte [representación de punto flotante IEEE](#).

Tipos enteros

El `int` tipo 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 utiliza de forma predeterminada o cuando la `signed` palabra clave Modifier está presente. La `unsigned` palabra clave Modifier 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 utilizada. El lenguaje admite `short` los `long` `long long` modificadores, y. Un `short` tipo debe tener al menos 16 bits de ancho. Un `long` tipo debe tener al menos 32 bits de ancho. Un `long long` tipo 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. Sin embargo, los tamaños reales pueden y varían entre las implementaciones. Vea [tamaños de tipos integrados](#) para obtener información detallada sobre la implementación específica de Microsoft.

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

Sinónimos de tipos enteros

El compilador considera los sinónimos de 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 `_int8` tipos, `_int16`, y de ancho específico `_int32` `_int64`. Estos tipos pueden usar los `signed` `unsigned` modificadores y. El tipo de datos `_int8` es sinónimo del tipo `char`, `_int16` es sinónimo del tipo `short`, `_int32` es sinónimo del tipo `int`, and `_int64` es sinónimo del tipo `long long`.

Tamaños de tipos integrados

La mayoría de los tipos integrados tienen tamaños definidos por la implementación. En la tabla siguiente se muestra la cantidad de almacenamiento necesario 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
<code>double</code> , <code>_int64</code> , <code>long double</code> , <code>long long</code> , <code>unsigned long long</code>	8 bytes

Vea [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, vea [conversiones estándar](#).

Consulta también

[Intervalos de tipo de datos](#)

Intervalos de tipo de datos

06/03/2021 • 4 minutes to read • [Edit Online](#)

Los compiladores de Microsoft C++ 32 y 64 bits reconocen los tipos de la tabla 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</code> , <code>short int</code> , <code>signed short int</code>	De -32 768 a 32 767
<code>unsigned _int16</code>	2	<code>unsigned short</code> , <code>unsigned short int</code>	De 0 a 65.535
<code>_int32</code>	4	<code>signed</code> , <code>signed int</code> , <code>int</code>	De -2.147.483.648 a 2.147.483.647
<code>unsigned _int32</code>	4	<code>unsigned</code> , <code>unsigned int</code>	De 0 a 4.294.967.295
<code>_int64</code>	8	<code>long long</code> , <code>signed long long</code>	De -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807

NOMBRE DEL TIPO	BYTES	OTROS NOMBRES	INTERVALO DE VALORES
<code>unsigned __int64</code>	8	<code>unsigned long long</code>	De 0 a 18.446.744.073.709.551.615
<code>bool</code>	1	ninguno	<code>false</code> o <code>true</code>
<code>char</code>	1	ninguno	de -128 a 127 de forma predeterminada de 0 a 255 cuando se compila con <code>/J</code>
<code>signed char</code>	1	ninguno	De -128 a 127
<code>unsigned char</code>	1	ninguno	De 0 a 255
<code>short</code>	2	<code>short int</code> , <code>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</code> , <code>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	ninguno	
<code>float</code>	4	ninguno	3,4E +/- 38 (7 dígitos)
<code>double</code>	8	ninguno	1,7E +/- 308 (15 dígitos)
<code>long double</code>	igual que <code>double</code>	ninguno	Igual que <code>double</code>
<code>wchar_t</code>	2	<code>_wchar_t</code>	De 0 a 65.535

Dependiendo de cómo se use, una variable de `_wchar_t` designa un tipo de caracteres anchos o un tipo de carácter 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 puede usar con cualquier tipo entero excepto `bool`. Tenga en cuenta que `char`, `signed char` y `unsigned char` son tres tipos distintos para los fines de mecanismos como la sobrecarga y las plantillas.

Los `int` `unsigned int` tipos y 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, vea [__int8, __int16, __int32, __int64](#) y [límites de enteros](#).

Para obtener más información sobre las restricciones de los tamaños de cada tipo, vea [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](#).

Consulta también

[Palabras clave](#)

[Tipos integrados](#)

nullptr

06/03/2021 • 2 minutes to read • [Edit Online](#)

La `nullptr` palabra clave especifica una constante de puntero NULL de tipo `std::nullptr_t`, que se pueden 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>`.

NOTE

La `nullptr` palabra clave también se define en C++/CLI para aplicaciones de código administrado y no es intercambiable con la palabra clave de C++ estándar de ISO. Si el código se puede compilar mediante la `/clr` opción del compilador, que tiene como destino el código administrado, use `__nullptr` en cualquier línea de código donde deba garantizar que el compilador usa la interpretación nativa de C++. Para obtener más información, vea [nullptr \(C++/CLI y C++/CX\)](#).

Observaciones

Evite usar `NULL` o cero (`0`) como una constante de puntero nulo; `nullptr` es menos vulnerable a un uso incorrecto y funciona mejor en la mayoría de los casos. 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 pueden convertir al `std::pair<const char *, double>` tipo de parámetro en `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>`.

Consulta también

Palabras clave

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

void (C++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

Cuando se usa como un tipo de valor devuelto de función, la `void` palabra clave 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 un puntero es `void * _`, *el puntero puede apuntar a cualquier variable que no esté declarada con la `const` o `volatile` palabra clave _ o.* No se puede desreferenciar un puntero `*void *_` a menos que se convierta en otro tipo. Un `puntero * void` se puede convertir en cualquier otro tipo de puntero de datos.

Un `void` puntero `_ *` puede apuntar a una función, pero no a un miembro de clase en C++.

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

Ejemplo

```
// void.cpp
void vobject;    // C2182
void *pv;        // okay
int *pint; int i;
int main() {
    pv = &i;
    // Cast optional in C required in C++
    pint = (int *)pv;
}
```

Consulte también

[Palabras clave](#)

[Tipos integrados](#)

bool (C++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 tanto, tienen valores de tipo `bool`. Por ejemplo, `i != 0` ahora tiene `true` o, `false` dependiendo del valor de `i`.

Visual Studio 2017 versión 15,3 y posterior (disponible con [/STD: c++ 17](#)): el operando de un operador de incremento o decremento de prefijo o de sufijo no puede ser de tipo `bool`. En otras palabras, dada una variable `b` de tipo `bool`, estas expresiones ya no se permiten:

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

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

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

En la instrucción siguiente:

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

Si `condexpr1` es `true`, `statement1` siempre se ejecuta; si `condexpr1` es `false`, `statement1` no se ejecuta nunca.

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

Visual Studio 2017 versión 15,3 y versiones posteriores: `operator++` para `bool` se ha quitado del lenguaje y ya no se admite.

`--` No se puede aplicar el operador de sufijo o de prefijo a una variable de este tipo.

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

Consulta también

[Palabras clave](#)

[Tipos integrados](#)

false (C++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 `true` expresión booleana). Por ejemplo, si `i` es una variable de tipo `bool`, la `i = false;` instrucción se asigna `false` a `i`.

Ejemplo

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

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

```
1
0
```

Consulte también

[Palabras clave](#)

true (C++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

Sintaxis

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

Observaciones

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 verdadera). Si `i` es de tipo `bool`, la instrucción `i = true;` asigna `true` a `i`.

Ejemplo

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

```
1
0
```

Consulte también

[Palabras clave](#)

char, wchar_t, char16_t, char32_t

06/03/2021 • 3 minutes to read • [Edit Online](#)

Los tipos `char`, `wchar_t`, `char16_t` y `char32_t` son tipos integrados que representan caracteres alfanuméricicos, así como glifos no alfanuméricos y caracteres no imprimibles.

Sintaxis

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

Observaciones

El `char` tipo era el tipo de carácter original en C y C++. El tipo `unsigned char` se usa a menudo para representar un *byte*, que no es un tipo integrado en C++. El `char` tipo se puede usar para almacenar caracteres del juego de caracteres ASCII o de cualquiera de los juegos de caracteres ISO-8859, así como de bytes individuales de caracteres de varios bytes, como Shift-JIS o la codificación UTF-8 del juego de caracteres Unicode. Las cadenas de `char` tipo se conocen como cadenas *estrechas*, incluso cuando se usan para codificar caracteres de varios bytes. En el compilador de Microsoft, `char` es un tipo de 8 bits.

El `wchar_t` tipo 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 la codificación Unicode codificada como UTF-16LE, el tipo de carácter nativo en los sistemas operativos Windows. Las versiones de caracteres anchos de las funciones de biblioteca en tiempo de ejecución universal C (UCRT) usan `wchar_t` y sus tipos de puntero y matriz como parámetros y valores devueltos, al igual que las versiones de caracteres anchos de la API nativa de Windows.

Los `char16_t` `char32_t` tipos y representan caracteres anchos de 16 y 32 bits, respectivamente. La codificación Unicode codificada como UTF-16 puede almacenarse en el `char16_t` tipo, y la codificación Unicode codificada como UTF-32 puede almacenarse en el `char32_t` tipo. Las cadenas de estos tipos y `wchar_t` se conocen como cadenas *anchas*, aunque el término suele referirse específicamente a cadenas de `wchar_t` tipo.

En la biblioteca estándar de C++, el `basic_string` tipo es especializado para cadenas anchas y estrechas. Use `std::string` cuando los caracteres sean de tipo `char`, `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` tienen especializaciones para cadenas estrechas y anchas.

`_int8, _int16, _int32, _int64`

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específico de Microsoft

Las características de Microsoft C/C++ admiten tipos enteros con tamaño. Puede declarar variables enteras de 8, 16, 32 o 64 bits mediante el `_intN` especificador de tipo, 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:

```
_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` y `_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 `_int8` tipo de datos es sinónimo del tipo `char`, `_int16` es sinónimo de tipo `short` y `_int32` es sinónimo de tipo `int`. El `_int64` tipo es sinónimo de tipo `long long`.

Por compatibilidad con versiones anteriores,, `_int8` `_int16` `_int32` y `_int64` son sinónimos de `_int8` , `_int16` , y, `_int32` `_int64` a menos que se especifique la opción del compilador [/za](#) (deshabilitar extensões de lenguaje).

Ejemplo

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

```
// 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
}
```

```
func
```

Consulta también

[Palabras clave](#)

[Tipos integrados](#)

[Intervalos de tipo de datos](#)

_m64

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

El `_m64` tipo de datos es para su uso con MMX y 3DNow! intrínsecos y se define en `<xmmmintrin.h>`.

```
// data_types_m64.cpp
#include <xmmmintrin.h>
int main()
{
    _m64 x;
}
```

Observaciones

No debe tener acceso directamente a los `_m64` campos. 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 `_m64` tipo de datos 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

Vea también

[Palabras clave](#)

[Tipos integrados](#)

[Intervalos de tipo de datos](#)

__m128

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

El tipo de **datos __m128**, para su uso con las instrucciones intrínsecas de extensiones SIMD de streaming y extensiones SIMD de streaming 2, se define en <xmmmintrin.h> .

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

Observaciones

No debe tener acceso directamente a los `__m128` campos. 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 `__m128` tipo de datos no es compatible con los procesadores ARM.

FIN de Específicos de Microsoft

Vea también

[Palabras clave](#)

[Tipos integrados](#)

[Intervalos de tipo de datos](#)

__m128d

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

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

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

Observaciones

No debe tener acceso directamente a los `__m128d` campos. 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 `__m128d` tipo de datos no es compatible con los procesadores ARM.

FIN de Específicos de Microsoft

Vea también

[Palabras clave](#)

[Tipos integrados](#)

[Intervalos de tipo de datos](#)

__m128i

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

El `__m128i` tipo de datos, para su uso con las instrucciones intrínsecas de extensiones SIMD de streaming 2 (sse2), se define en `<emmintrin.h>`.

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

Observaciones

No debe tener acceso directamente a los `__m128i` campos. 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.

NOTE

El uso de variables de tipo hará `__m128i` que el compilador genere la `movdqa` instrucción SSE2. Esta instrucción no provoca un error en procesadores Pentium III, pero producirá un error silencioso, con posibles efectos secundarios causados por las instrucciones que se `movdqa` traducen en procesadores Pentium III.

El `__m128i` tipo de datos no es compatible con los procesadores ARM.

FIN de Específicos de Microsoft

Vea también

[Palabras clave](#)

[Tipos integrados](#)

[Intervalos de tipo de datos](#)

__ptr32, __ptr64

06/03/2021 • 2 minutes to read • [Edit Online](#)

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:

```
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.

NOTE

No se puede usar `__ptr32` o `__ptr64` cuando se compila con `/clr: Pure`. De lo contrario, se generará el error del compilador C2472. Las opciones del compilador `/clr: Pure` y `/clr: Safe` están en desuso en Visual Studio 2015 y no se admiten en Visual Studio 2017.

Por compatibilidad con versiones anteriores, `__ptr32` y `__ptr64` son sinónimos para `__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 `__ptr32` las `__ptr64` palabras clave y.

```
#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;
}
```

32
64

Vea también

[Tipos integrados](#)

Límites numéricos (C++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

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

Consulta también

[Conceptos básicos](#)

Límites de enteros

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 al incluir el archivo de encabezado estándar <climits> .

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 <code>/J</code> usa la opción
<code>CHAR_MAX</code>	Valor máximo de una variable de tipo <code>char</code> .	127; 255 si se <code>/J</code> usa la opción
<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)

CONSTANTE	SIGNIFICADO	VALOR
<code>LONG_MIN</code>	Valor mínimo de una variable de tipo <code>long</code> .	-2147483648
<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 (0xfffffffffffffffff)

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

Consulta también

[Límites flotantes](#)

Límites flotantes

06/03/2021 • 2 minutes to read • [Edit Online](#)

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> <code>DBL_DIG</code> <code>LDBL_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 se puede restablecer sin pérdida de precisión.	6 15 15
<code>FLT_EPSILON</code> <code>DBL_EPSILON</code> <code>LDBL_EPSILON</code>	Número positivo menor x , tal que $x + 1,0$ no es igual a $1,0$.	1.192092896e-07F 2.2204460492503131e-016 2.2204460492503131e-016
<code>FLT_GUARD</code>		0
<code>FLT_MANT_DIG</code> <code>DBL_MANT_DIG</code> <code>LDBL_MANT_DIG</code>	Número de dígitos en la base especificada por <code>FLT_RADIX</code> en el significado de punto flotante. La base es 2; por lo tanto, estos valores especifican bits.	24 53 53
<code>FLT_MAX</code> <code>DBL_MAX</code> <code>LDBL_MAX</code>	Número de punto flotante máximo que se va a representar.	3.402823466e+38F 1.7976931348623158e+308 1.7976931348623158e+308
<code>FLT_MAX_10_EXP</code> <code>DBL_MAX_10_EXP</code> <code>LDBL_MAX_10_EXP</code>	Entero máximo tal que 10 elevado a dicho número es un número de punto flotante que se pueda representar.	38 308 308
<code>FLT_MAX_EXP</code> <code>DBL_MAX_EXP</code> <code>LDBL_MAX_EXP</code>	Entero máximo tal que <code>FLT_RADIX</code> se eleva a ese número es un número de punto flotante que se pueda representar.	128 1024 1024
<code>FLT_MIN</code> <code>DBL_MIN</code> <code>LDBL_MIN</code>	Valor positivo mínimo.	1.175494351e-38F 2.2250738585072014e-308 2.2250738585072014e-308
<code>FLT_MIN_10_EXP</code> <code>DBL_MIN_10_EXP</code> <code>LDBL_MIN_10_EXP</code>	Entero negativo mínimo tal que 10 elevado a dicho número es un número de punto flotante que se pueda representar.	-37 -307 -307

CONSTANTE	SIGNIFICADO	VALOR
<code>FLT_MIN_EXP</code> <code>DBL_MIN_EXP</code> <code>LDBL_MIN_EXP</code>	Entero negativo mínimo tal que <code>FLT_RADIX</code> se eleva a ese número es un número de punto flotante que se pueda representar.	-125 -1021 -1021
<code>FLT_NORMALIZE</code>		0
<code>FLT_RADIX</code> <code>_DBL_RADIX</code> <code>_LDBL_RADIX</code>	Base de representación de exponente.	2 2 2
<code>FLT_ROUNDS</code> <code>_DBL_ROUNDS</code> <code>_LDBL_ROUNDS</code>	Modo de redondeo para la adición de punto flotante.	1 (próximo) 1 (próximo) 1 (próximo)

NOTE

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

FIN de Específicos de Microsoft

Vea también

[Límites de enteros](#)

Declaraciones y definiciones (C++)

06/03/2021 • 9 minutes to read • [Edit Online](#)

Un programa de C++ consta de varias entidades, como variables, funciones, tipos y espacios de nombres. Cada una de estas entidades debe *declararse* antes de que se puedan utilizar. 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 hace visible para el compilador. No se puede hacer referencia a una función o clase declarada en algún punto posterior de la unidad de compilación. Las variables deben declararse lo más cerca posible antes del punto en el que se usan.

En el ejemplo siguiente se muestran algunas declaraciones:

```
#include <string>

void f(); // forward declaration

int main()
{
    const double pi = 3.14; //OK
    int i = f(2); //OK. f is forward-declared
    std::string str; // OK std::string is declared in <string> header
    C obj; // error! C not yet declared.
    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, `main` se declara la función. En la línea 7, `const pi` se declara e *inicializa* una variable denominada. En la línea 8, `i` se declara un entero y se inicializa con el valor generado por la función `f`. El nombre `f` es visible para el compilador debido a la *declaración adelantada* 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 hasta después en el programa y no se declara hacia delante. Para corregir el error, puede desplazar la *definición* completa de `C` antes `main` o agregar una declaración de avance para ella. Este comportamiento es diferente de otros lenguajes como C#, en el que se pueden usar funciones y clases antes de su punto de declaración en un archivo de código fuente.

En la línea 10, se declara una variable denominada `str` de tipo `std::string`. El nombre `std::string` es visible porque se presenta en el `string` *archivo de encabezado* que se combina en el archivo de código fuente en la línea 1. `std` es el espacio de nombres en el que `string` se declara la clase.

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

Ámbito de la declaración

El nombre que se introduce en una declaración es válido dentro del *ámbito* en el que se produce la declaración. En el ejemplo anterior, las variables que se declaran dentro de la `main` función son *variables locales*. Podría declarar otra variable denominada `i` fuera de Main, en el *ámbito global*, y sería una entidad completamente independiente. Sin embargo, dicha duplicación de nombres puede conducir a errores y confusión 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 *conflictos de nombres*. La mayoría de los nombres de biblioteca estándar de C++ se declaran dentro del `std` espacio de nombres. Para obtener más información sobre cómo interactúan las reglas de ámbito con las declaraciones, vea [ámbito](#).

Definiciones

Algunas entidades, incluidas las funciones, las clases, las enumeraciones y las variables constantes, deben definirse además de declararse. Una *definición* proporciona al compilador toda la información que necesita para generar código 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 se define (aunque, tal y como se define, la clase no hace nada). Se debe definir una variable constante, en otras palabras asignadas a un valor, en la misma instrucción en la que se declara. Una declaración de un tipo integrado como `int` es automáticamente una definición, ya que el compilador conoce la cantidad de espacio que se va a asignar.

En el ejemplo siguiente se muestran las declaraciones que también son definiciones:

```
// 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:

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

Typedefs y instrucciones Using

En versiones anteriores de C++, la `typedef` palabra clave se utiliza para declarar un nuevo nombre que es un *alias* de 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 la definición de tipo y no el nombre real. En C++ moderno, la `using` palabra clave es preferible `typedef`, pero la idea es la misma: se ha declarado un nuevo nombre para una entidad que ya se ha declarado y definido.

Miembros de clase estáticos

Dado que los miembros de datos de clase estática son variables discretas compartidas por todos los objetos de

la clase, deben definirse e inicializarse fuera de la definición de clase. (Para obtener más información, vea [clases](#)).

declaraciones extern

Un programa de C++ puede contener más de una [unidad de compilación](#). Para declarar una entidad que se define en una unidad de compilación independiente, utilice la `extern` palabra clave. La información de la declaración es suficiente para el compilador, pero si la definición de la entidad no se encuentra en el paso de vinculación, el enlazador producirá un error.

En esta sección

Clases de almacenamiento

`const`

`constexpr`

`extern`

Inicializadores

Alias y definiciones de tipo

`using` relativa

`volatile`

`decltype`

Atributos en C++

Consulta también

Conceptos básicos

Clases de almacenamiento

02/11/2020 • 14 minutes to read • [Edit Online](#)

Una *clase de almacenamiento* en el contexto de las declaraciones de variables de C++ es un especificador de tipo que rige la duración, la vinculación y la ubicación de la memoria de los 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 mediante los `extern`, `static` y `thread_local` especificadores, o. Las variables y objetos automáticos no tienen ninguna vinculación; no son visibles para código fuera del bloque. La memoria se asigna automáticamente cuando la ejecución entra en el bloque y se anula la asignación cuando se sale del bloque.

Notas

1. 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.
2. **Visual Studio 2010 y versiones posteriores:** La `auto` palabra clave ya no es un especificador de clase de almacenamiento de C++ y la `register` palabra clave está en desuso. **Visual Studio 2017 versión 15,7 y versiones posteriores:** (disponible con `/std:c++17`): la `register` palabra clave se quita del lenguaje C++.

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

static

La `static` palabra clave se puede utilizar 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 `static` palabra clave se puede usar en las siguientes situaciones.

1. Cuando se declara una variable o función en el ámbito de archivo (ámbito global o de espacio de nombres), la `static` palabra clave especifica que la variable o función tiene 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 `static` palabra clave especifica que la variable mantiene su estado entre las llamadas a esa función.
3. Cuando se declara un miembro de datos en una declaración de clase, la `static` palabra clave especifica que todas las instancias de la clase comparten una copia del miembro. Un miembro de datos estático 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 `static` palabra clave especifica que todas las instancias de la clase comparten la función. Una función miembro estática no puede tener

acceso a un miembro de instancia porque la función no tiene un `this` puntero implícito. Para tener acceso a un miembro de instancia, declare la función con un parámetro que sea un puntero o referencia de instancia.

5. No puede declarar los miembros de una unión como estáticos. Sin embargo, se debe declarar explícitamente una Unión anónima declarada globalmente `static`.

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

```
// 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 );
}
```

```
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.

```
// 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;
}
```

```
0
0
1
1
2
2
3
3
```

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

```
// 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);
}
```

```
var != value
var == value
```

A partir de C++ 11, `static` se garantiza que la inicialización de una variable local es segura para subprocessos. Esta característica se denomina a veces *estáticas mágicas*. Sin embargo, en una aplicación con subprocessamiento múltiple todas las asignaciones posteriores deben estar sincronizadas. La característica de inicialización estática segura para subprocessos se puede deshabilitar mediante la `/Zc:threadSafeInit-` marca para evitar la realización de una dependencia en CRT.

extern

Los objetos y las variables que se declaran como declaran `extern` un objeto que se define en otra unidad de traducción o en un ámbito de inclusión como una vinculación externa. Para obtener más información, consulte [extern](#) y [unidades de traducción y vinculación](#).

thread_local (C++ 11)

Una variable declarada con el `thread_local` especificador 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 específico de Microsoft `__declspec(thread)`.

```

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 que se deben tener en cuenta sobre el `thread_local` especificador:

- Es posible que las variables locales de subprocesso inicializadas dinámicamente en archivos dll no se inicialicen correctamente en todos los subprocessos de llamada. Para obtener más información, vea [thread](#).
- El `thread_local` especificador se puede combinar con `static` o `extern`.
- Solo se puede aplicar `thread_local` a declaraciones y definiciones de datos; `thread_local` no se puede usar en declaraciones o definiciones de función.
- Solo se puede especificar `thread_local` en elementos de datos con duración de almacenamiento estática. Esto incluye los objetos de datos globales (`static` y `extern`), los objetos estáticos locales y los miembros de datos estáticos de las 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 subprocesso, ya sea que la declaración y la definición se produzcan en el mismo archivo o en archivos independientes.

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

el

Visual Studio 2017 versión 15,3 y posterior (disponible con `/std:c++17`): la `register` palabra clave ya no es una clase de almacenamiento admitida. La palabra clave todavía está reservada en el estándar para su uso futuro.

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

Ejemplo: inicialización automática frente a inicialización 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`:

```

// 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";
}

```

```

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 **I1** se **I2** inicializan los objetos, y **I3** cuando se destruyen.

Hay varios puntos que se deben tener en cuenta 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). El resultado se muestra exactamente cuando se inicializan.
- Por último, las variables locales estáticas como `I3` conservan sus valores mientras dura el programa, pero se destruyen cuando el programa finaliza.

Consulte también

[Declaraciones y definiciones](#)

auto (C++)

06/03/2021 • 12 minutes to read • [Edit Online](#)

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

NOTE

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

Sintaxis

```
auto ***inicializador* *declarador*** ;
```

```
[](auto ***parám1* , auto *parám2*** ) {};
```

Observaciones

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

Se recomienda usar la `auto` palabra clave para la mayoría de las situaciones, a menos que desee realmente una conversión, ya que proporciona estas ventajas:

- **Solidez:** Si se cambia el tipo de la expresión (esto incluye cuando se cambia un tipo de valor devuelto de función), solo funciona.
- **Rendimiento:** Se garantiza que no habrá conversión.
- **Facilidad de uso:** No tiene que preocuparse por las dificultades de ortografía y los errores tipográficos.
- **Eficiencia:** La codificación puede ser más eficaz.

Casos de conversión en los que es posible que no desee usar `auto`:

- Cuando se desea un tipo específico y es la única alternativa.
- Tipos de aplicación auxiliar de plantilla de expresión: por ejemplo, `(valarray+valarray)`.

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

La `auto` expresión de inicialización puede adoptar varias formas:

- Sintaxis de inicialización universal, como `auto a { 42 };`.
- Sintaxis de asignación, como `auto b = 0;`.

- Sintaxis de asignación universal, que combina los dos formularios anteriores, como `auto c = { 3.14156 };` .
- Inicialización directa o sintaxis de tipo 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 utiliza para declarar el parámetro de bucle en una instrucción basada en intervalo `for`, se usa una sintaxis de inicialización diferente, por ejemplo `for (auto& i : iterable) do_action(i);`. Para obtener más información, vea [instrucción basada en intervalo `for` \(C++\)](#).

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

Utilidad

La `auto` palabra clave es una forma sencilla de declarar una variable que tiene un tipo complejo. Por ejemplo, puede utilizar `auto` para declarar una variable en la que la expresión de inicialización implique plantillas, punteros a funciones o punteros a miembros.

También 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

Puede usar `auto`, junto con el `decltype` especificador de tipo, para ayudar a escribir bibliotecas de plantillas. Utilice `auto` y `decltype` para declarar una función de plantilla cuyo tipo de valor devuelto depende de los tipos de sus argumentos de plantilla. O bien, use `auto` y `decltype` para declarar una función de plantilla que contenga una llamada a otra función y, a continuación, devuelva el tipo de valor devuelto de esa otra función. Para obtener más información, vea [decltype](#).

Referencias y calificadores cv

Tenga en cuenta que el uso de `auto` gotas de referencias, `const` calificadores y `volatile` calificadores.

Considere el ejemplo siguiente:

```
// 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, he auto 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 se ha quitado `auto`.

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

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

```
#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 del uso de la `auto` palabra clave y el mensaje de error de diagnóstico correspondiente que emite el compilador.

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 <code>auto</code> palabra clave debe tener un inicializador.
C3532	Ha utilizado incorrectamente la <code>auto</code> palabra clave para declarar un tipo. Por ejemplo, declaró un tipo de valor devuelto de método o una matriz.
C3533, C3539	Un parámetro o un argumento de plantilla no se pueden declarar con la <code>auto</code> palabra clave.
C3535	Un método o un parámetro de plantilla no se puede declarar con la <code>auto</code> palabra clave.
C3536	No se puede usar un símbolo antes de inicializarlo. En la práctica, esto significa que una variable no se puede usar para inicializarse a sí misma.
C3537	No se puede convertir a un tipo que se declara con la <code>auto</code> palabra clave.

NÚMERO DE ERROR	DESCRIPCIÓN
C3538	Todos los símbolos de una lista de declaradores que se declara con la <code>auto</code> palabra clave 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 <code>auto</code> palabra clave.

Ejemplos

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

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

```
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. Uno de los motivos más atractivos para usar la `auto` palabra clave es la simplicidad.

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

En el fragmento de código siguiente se declara el tipo de variables `iter` y `elem`. Cuándo se `for` `for` inician los bucles de intervalos y.

```
// 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 `new` se usa el operador y la declaración de puntero para declarar punteros.

```
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.

```
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:

```
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 `fp` la variable en un puntero a una función que devuelve el tipo `int`.

```
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` Operator

`typeid`

`operator new`

Declaraciones y definiciones

Ejemplos de expresiones lambda

Inicializadores

`decltype`

const (C++)

06/03/2021 • 6 minutes to read • [Edit Online](#)

Al modificar una declaración de datos, la `const` palabra clave especifica que el objeto o la variable no es modificable.

Sintaxis

```
const declaration ;
member-function const ;
```

valores const

La `const` palabra clave especifica que el valor de una variable es constante e indica al compilador que evite que el programador la modifique.

```
// constant_values1.cpp
int main() {
    const int i = 5;
    i = 10;    // C3892
    i++;      // C2105
}
```

En C++, puede usar la `const` palabra clave en lugar de la Directiva de preprocessador `#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 `const` variable como se indica a continuación:

```
// 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 `const` palabra clave también se puede utilizar en declaraciones de puntero.

```
// constant_values3.cpp
int main() {
    char *mybuf = 0, *yourbuf;
    char *const aptr = mybuf;
    *aptr = 'a';    // OK
    aptr = yourbuf; // C3892
}
```

Un puntero a una variable declarada como `const` solo se puede asignar a un puntero que también se declara como `const`.

```
// 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.

En el caso de los objetos que se declaran como `const`, solo puede llamar a funciones miembro de constante. Esto garantiza que el objeto constante nunca se modifique.

```
birthday.getMonth(); // Okay
birthday.setMonth( 4 ); // Error
```

Se puede llamar a funciones miembro de constante o que no son de constante para un objeto que no es constante. También puede sobrecargar una función miembro mediante la `const` palabra clave; esto permite que se llame a una versión diferente de la función para objetos constantes y no constantes.

Los constructores o destructores no se pueden declarar con la `const` palabra clave.

funciones miembro const

Al declarar una función miembro con la `const` palabra clave, se especifica 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 constantes. Para declarar una función miembro constante, coloque la `const` palabra clave después del paréntesis de cierre de la lista de argumentos. La `const` palabra clave es necesaria tanto en la declaración como en la definición.

```

// 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 C y C++ const

Cuando se declara una variable como `const` en un archivo de código fuente de C, se hace así:

```
const int i = 2;
```

A continuación, esta variable se puede utilizar en otro módulo como sigue:

```
extern const int i;
```

Pero para obtener el mismo comportamiento en C++, debe declarar la `const` variable como:

```
extern const int i = 2;
```

Si desea declarar una `extern` variable en un archivo de código fuente de C++ para su uso en un archivo de código fuente de C, use:

```
extern "C" const int x=10;
```

para evitar que el compilador de C++ elimine nombres.

Observaciones

Cuando se sigue la lista de parámetros de una función miembro, la `const` palabra clave especifica que la función no modifica el objeto para el que se invoca.

Para obtener más información sobre `const`, vea los temas siguientes:

- Punteros const y volatile
- Calificadores de tipo (Referencia del lenguaje C)
- volatile
- #define

Consulta también

Palabras clave

constexpr (C++)

02/11/2020 • 9 minutes to read • [Edit Online](#)

La palabra clave `constexpr` se presentó en C++ 11 y se mejoró en C++ 14. Significa * const expresión ANT*.

Como `const`, se puede aplicar a las variables: se genera un error del compilador cuando cualquier código intenta modificar `if` y el valor. A diferencia de `const`, `constexpr` también se puede aplicar a las funciones y a la clase `const ructors`. `constexpr` indica que el valor, o el valor devuelto, es const ANT y, siempre que sea posible, se calcula en tiempo de compilación.

Se `constexpr` puede utilizar un valor entero siempre const que se requiera un entero, como en los argumentos de plantilla y en las declaraciones de matriz. Y cuando un valor se calcula en tiempo de compilación en lugar de en tiempo de ejecución, ayuda a que el programa se ejecute más rápido y use menos memoria.

Para limitar la complejidad de los const cálculos ANT en tiempo de compilación y sus posibles impactos en el tiempo de compilación, el estándar de C++ 14 requiere que los tipos de las const expresiones ANT sean [tipos literales](#).

Sintaxis

```
constexpr expresión de tipo literal/* if IEr* = * const ANT-Expression* ;  
constexpr * if IEr* de tipo literal{ * const ANT-Expression* } ;  
constexpr * if IEr* de tipo literal( params ) ;  
* constexpr ***ctor( params ) ;
```

Parámetros

params

Uno o más parámetros, cada uno de los cuales debe ser un tipo literal y debe ser una const expresión Ant.

Valor devuelto

Una `constexpr` variable o función debe devolver un [tipo literal](#).

Variables `constexpr`

La deducción principal `const` entre `constexpr` las variables y es que la inicialización de una `const` variable se puede diferir hasta el tiempo de ejecución. Una `constexpr` variable debe inicializarse en tiempo de compilación. Todas `constexpr` las variables son `const`.

- Una variable se puede declarar con `constexpr`, cuando tiene un tipo literal y se inicializa. Si la inicialización es por medio de `const ructor`, el `const ructor` debe declararse como `constexpr`.
- Una referencia se puede declarar como `constexpr` cuando se cumplen ambas condiciones: el objeto al que se hace referencia se inicializa con una const expresión ANT y cualquier conversión implícita invocada durante la inicialización también son const expresiones de Ant.
- Todas las declaraciones de una `constexpr` variable o función deben tener la `constexpr` especificación `if IEr`.

```

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

```

Funciones constexpr

Una `constexpr` función es aquella cuyo valor devuelto se calcula en tiempo de compilación cuando el código utilizado lo requiere. El código de consumo requiere el valor devuelto en tiempo de compilación para inicializar una `constexpr` variable o para proporcionar un argumento de plantilla sin tipo. Cuando sus argumentos son `constexpr` valores, una `constexpr` función genera un tiempo de compilación const Ant. Cuando se llama con argumentos que no son de `constexpr`, o cuando su valor no es necesario en tiempo de compilación, genera un valor en tiempo de ejecución como una función normal. (Este comportamiento dual evita tener que escribir `constexpr` y no `constexpr` versiones de la misma función).

Una `constexpr` función o const ructor es implícitamente `inline`.

Las siguientes reglas se aplican a `constexpr` las funciones:

- Una `constexpr` función debe aceptar y devolver solo [tipos literales](#).
- Una `constexpr` función puede ser recursiva.
- No puede ser [virtual](#). constNo se puede definir un ructor como `constexpr` cuando la clase envolvente tiene clases base virtuales.
- El cuerpo se puede definir como `= default` o `= delete`.
- El cuerpo no puede contener `goto` instrucciones ni `try` bloques.
- Una especialización explícita de una plantilla que no es de `constexpr` se puede declarar como `constexpr`:
- Una especialización explícita de una `constexpr` plantilla tampoco tiene que ser `constexpr`:

Las siguientes reglas se aplican a `constexpr` las funciones de Visual Studio 2017 y versiones posteriores:

- Puede contener `if` instrucciones y `switch`, y todas las instrucciones de bucle, incluidas `for`, basadas en intervalos `for`, `while` y ** `while` **.
- Puede contener declaraciones de variables locales, pero se debe inicializar la variable. Debe ser un tipo literal y no puede ser `static` o local del subprocesso. No es necesario que la variable declarada localmente sea `const` y puede mutar.
- `constexpr static` No es necesario que una función no miembro sea implícita `const`.

```

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;
}

```

TIP

En el depurador de Visual Studio, al depurar una compilación de depuración no optimizada, puede saber si una `constexpr` función se evalúa en tiempo de compilación colocando un punto de interrupción dentro de ella. 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.

externas constexpr

La opción del compilador `/Zc: externConstexpr` hace que el compilador aplique una [vinculación externa](#) a las variables declaradas mediante `extern constexpr`. En versiones anteriores de Visual Studio, de forma predeterminada o cuando `/Zc: externConstexpr-es` la especificación de la configuración de if la aplicación, Visual Studio aplica la vinculación interna a `constexpr` las variables incluso cuando `extern` se usa la palabra clave. 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 ejemplo siguiente se muestran `constexpr` variables, funciones y un tipo definido por el usuario. En la última instrucción de `main()`, la `constexpr` función miembro `GetValue()` es una llamada en tiempo de ejecución porque no es necesario que el valor sea conocido en tiempo de compilación.

```

// 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.

Vea también

[Declaraciones y definiciones](#)

[const](#)

extern (C++)

06/03/2021 • 7 minutes to read • [Edit Online](#)

La `extern` palabra clave se puede aplicar a una declaración de variable, función o plantilla global. Especifica que el símbolo tiene la * extern vinculación*. Para obtener información general sobre la vinculación y el motivo por el que no se recomienda el uso de variables globales, vea [unidades de traducción y vinculación](#).

La `extern` palabra clave tiene cuatro significados según el contexto:

- En una declaración de `const` variable no global, `extern` especifica que la variable o función se define en otra unidad de traducción. Se `extern` debe aplicar en todos los archivos excepto en el que se define la variable.
- En una `const` declaración de variable, especifica que la variable tiene la extern vinculación. `extern` Se debe aplicar a todas las declaraciones de todos los archivos. (`const` Las variables globales tienen vinculación interna de forma predeterminada).
- ** `extern "C"** especifica que la función se define en otra parte y usa la Convención de llamada del lenguaje C. El extern modificador "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 volver a usar la otra instancia, en lugar de crear una nueva en la ubicación actual. Para obtener más información sobre este uso de `extern`, vea [creación de instancias explícitas](#).

externvinculación de no const globales

Cuando el vinculador ve `extern` antes de una declaración de variable global, busca la definición en otra unidad de traducción. Las declaraciones de variables que no son `const` variables en el ámbito global son extern al de forma predeterminada. Solo `extern` se aplica a las declaraciones que no proporcionan la definición.

```
//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)
```

externvinculación para const variables globales

De `const` forma predeterminada, una variable global tiene vinculación interna. Si desea que la variable tenga extern vinculación, aplique la `extern` palabra clave a la definición y a todas las demás declaraciones de otros archivos:

```
//fileA.cpp
extern const int i = 42; // extern const definition

//fileB.cpp
extern const int i; // declaration only. same as i in FileA
```

externconstexpr vinculación

En la versión 15,3 y anteriores de Visual Studio 2017, el compilador siempre dio una `constexpr` vinculación interna variable, incluso cuando la variable se marcó `extern`. En la versión 15,5 de Visual Studio 2017 y versiones posteriores, el modificador de compilador `/Zc: extern Constexpr` permite un comportamiento correcto que cumple con los estándares. Finalmente, la opción se convertirá en el valor predeterminado. La `/permissive-` opción no habilita `/Zc: extern Constexpr`.

```
extern constexpr int x = 10; //error LNK2005: "int const x" already defined
```

Si un archivo de encabezado contiene una variable declarada `extern constexpr`, se debe marcar para que se `__declspec(selectany)` combinen correctamente sus declaraciones duplicadas:

```
extern constexpr __declspec(selectany) int x = 10;
```

externDeclaraciones de función "C" y extern "C++"

En C++, cuando se usa con una cadena, `extern` especifica que se usan las convenciones de vinculación de otro lenguaje para los declaradores. Solo se puede tener acceso a las funciones y los datos de C si se han declarado previamente como una 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 *literal de cadena*. Todos los archivos de inclusión estándar utilizan la sintaxis `** extern "C"**` para permitir que las funciones de la biblioteca en tiempo de ejecución se utilicen en programas de C++.

Ejemplo

En el ejemplo siguiente se muestra cómo declarar nombres que tienen vinculación 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, deben coincidir. Es un error declarar funciones como una 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:

```

extern "C" int CFunc1();
...
int CFunc1();           // Redefinition is benign; C linkage is
                       // retained.

int CFunc2();
...
extern "C" int CFunc2(); // Error: not the first declaration of
                       // CFunc2; cannot contain linkage
                       // specifier.

```

Consulte también

[Palabra](#)

[Unidades de traducción y vinculación](#)

[externEspecificador de clase de almacenamiento en C](#)

[Comportamiento de los identificadores en C](#)

[Vinculación en C](#)

Inicializadores

02/11/2020 • 20 minutes to read • [Edit Online](#)

Un inicializador especifica el valor inicial de una variable. Se pueden inicializar variables en estos contextos:

- En la definición de una variable:

```
int i = 3;
Point p1{ 1, 2 };
```

- Como uno de los parámetros de una función:

```
set_point(Point{ 5, 6 });
```

- Como el valor devuelto de una función:

```
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:

```
Point p1(1, 2);
```

- Un signo igual seguido de una expresión:

```
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:

```
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({}); // empty list
    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 .
- Las matrices, las clases Pod , las estructuras y las uniones tienen sus miembros inicializados 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:

```
struct my_struct{
    int i;
    char c;
};

int i0;           // zero-initialized to 0
int main() {
    static float f1; // zero-initialized to 0.00000000
    double d{};     // zero-initialized to 0.0000000000000000
    int* ptr{};     // initialized to nullptr
    char s_array[3]{'a', 'b'}; // the third char is initialized to '\0'
    int int_array[5] = { 8, 9, 10 }; // the fourth and fifth ints are initialized to 0
    my_struct a_struct{}; // i = 0, c = '\0'
}
```

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 new palabra clave:

```
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.

```
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:

```
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:

```
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):

```
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 acerca de la inicialización de objetos estáticos globales, vea [la función Main y los argumentos 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 `new` palabra clave más 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

```
class BaseClass {  
private:  
    int m_int;  
};  
  
int main() {  
    BaseClass bc{};      // class is initialized  
    BaseClass* bc2 = new BaseClass(); // class is initialized, m_int value is 0  
    int int_arr[3]{};   // value of all members is 0  
    int a{};          // value of a is 0  
    double b{};        // value of b is 0.0000000000000000  
}
```

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 Vea [inicialización de agregado](#) para obtener ejemplos.

En el código siguiente se muestran varios ejemplos de inicialización de copia:

```

#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:

```

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 `new` palabra clave 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:

```

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
- una clase se inicializa con la `new` palabra clave
- 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:

```

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

NOTE

En Visual Studio 2015 y versiones anteriores, no se permite que un agregado tenga inicializadores de llave o igualdad para los miembros no estáticos. Esta restricción se ha quitado en el estándar de C++ 14 y se ha implementado 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:

```

#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 ver la siguiente salida:

```

agg1: c: 1
agg2: Z: 2
myArr1: 1 2 3 4
myArr3: 8 9 10 0 0

```

IMPORTANT

Los miembros de la matriz que se declaran pero no se inicializan explícitamente durante la inicialización de agregado se inicializan en cero, como en el caso `myArr3` anterior.

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:

```

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:

```

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:

```

// 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:

```
int func( int& );
```

- Declaraciones de tipo de valor devuelto de función. Por ejemplo:

```
int& func( int& );
```

- Declaración de un miembro de clase de tipo de referencia. Por ejemplo:

```
class c {public:    int& i;};
```

- Declaración de una variable especificada explícitamente como `extern`. Por ejemplo:

```
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.

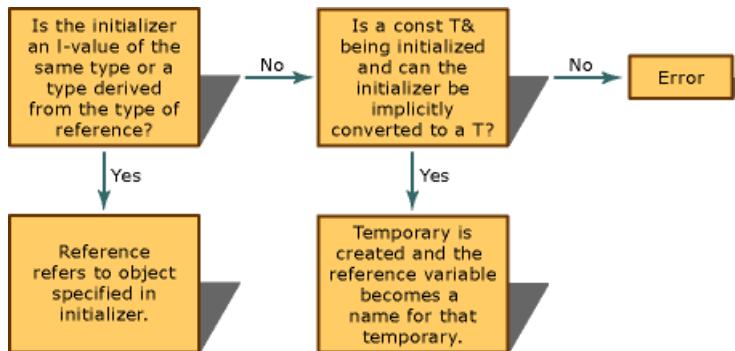


Gráfico de decisión para la inicialización de tipos de referencia

Las referencias a `volatile` tipos (declarados como `volatile typename & identificadorTypeName`) se pueden inicializar con `volatile` objetos del mismo tipo o con objetos que no se han declarado como `volatile`. No se pueden inicializar con `const` objetos de ese tipo. De forma similar, las referencias a `const` tipos (declarados como `const typename & identificadorTypeName`) se pueden inicializar con `const` objetos del mismo tipo (o con cualquier objeto que tenga una conversión a ese tipo) o con objetos que no se hayan declarado como `const`. No se pueden inicializar con `volatile` objetos de ese tipo.

Las referencias que no están calificadas con `const` la `volatile` palabra clave o solo se pueden inicializar con

objetos declarados como `const` ni `volatile` .

Inicialización de variables externas

Las declaraciones de variables automáticas, estáticas y externas pueden contener inicializadores. Sin embargo, las declaraciones de variables externas solo pueden contener inicializadores si las variables no se declaran como `extern` .

Alias y definiciones de tipos (C++)

06/03/2021 • 11 minutes to read • [Edit Online](#)

Puede usar una *declaración de alias* para declarar un nombre que se usará como sinónimo de un tipo declarado previamente. (Este mecanismo también se denomina informativo como un *alias de tipo*). También puede utilizar este mecanismo para crear una *plantilla de alias*, que puede ser especialmente útil para los asignadores personalizados.

Sintaxis

```
using identifier = type;
```

Observaciones

identifier

Nombre del alias.

type

Identificador de tipo para el que se va a crear un alias.

Un alias no presenta un tipo nuevo y no puede cambiar el significado de un nombre de tipo existente.

La forma más sencilla de un alias es equivalente al `typedef` mecanismo de c++ 03:

```
// C++11
using counter = long;

// C++03 equivalent:
// typedef long counter;
```

Ambos permiten la creación de variables de tipo "contador". Algo más útil sería un alias de tipo como este para

```
std::ios_base::fmtflags :
```

```
// 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++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 `typedef` mecanismo es que no funciona con plantillas. Sin embargo, la sintaxis de alias de tipo de C++11 permite la creación de plantillas de alias:

```
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 para 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.

```

#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;
}

```

1701 1764 1664

Typedefs

Una `typedef` declaración introduce un nombre que, dentro de su ámbito, se convierte en un sinónimo del tipo dado por la parte de la declaración de *tipo* de la declaración.

Puede utilizar declaraciones `typedef` para construir nombres más cortos o más significativos para tipos ya definidos por el lenguaje o para tipos que ha declarado. Los nombres de `typedef` permiten encapsular detalles de la implementación que pueden cambiar.

A diferencia de las `class`, `struct`, `declaraciones`, `union` y `enum`, las `typedef` declaraciones no introducen

nuevos tipos; introducen nuevos nombres para los tipos existentes.

Los nombres declarados con `typedef` ocupan el mismo espacio de nombres que otros identificadores (excepto las etiquetas de instrucción). Por consiguiente, no pueden utilizar el mismo identificador que un nombre declarado previamente, excepto en una declaración de tipo de clase. Considere el ejemplo siguiente:

```
// typedef_names1.cpp
// C2377 expected
typedef unsigned long UL;    // Declare a typedef name, UL.
int UL;                      // C2377: redefined.
```

Las reglas de ocultación de nombres que pertenecen a otros identificadores también rigen la visibilidad de los nombres declarados mediante `typedef`. Por consiguiente, el ejemplo siguiente es válido en 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
```

```
// typedef_specifier1.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:

```
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:

```
const int FlagType; // Type specifier required
```

No basta con decir

```
const FlagType; // Incomplete specification
```

porque se considera que `FlagType` forma parte del tipo, no un identificador que se va a volver a declarar. Esta declaración se considera no válida como

```
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 `typedef` declaraciones es hacer que las declaraciones sean más uniformes y compactas. Por ejemplo:

```
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` para especificar tipos fundamentales y derivados en la misma declaración, puede separar los declaradores con comas. Por ejemplo:

```
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`:

```
typedef void DRAWF( int, int );
```

Después de la `typedef` instrucción anterior, la declaración

```
DRAWF box;
```

sería equivalente a la declaración

```
void box( int, int );
```

`typedef` se suele combinar con `struct` para declarar y nombrar tipos definidos por el usuario:

```
// 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);
}
```

Nueva declaración de definiciones de tipos

La `typedef` declaración se puede usar para volver a declarar el mismo nombre para hacer referencia al mismo tipo. Por ejemplo:

```
// FILE1.H
typedef char CHAR;

// FILE2.H
typedef char CHAR;

// PROG.CPP
#include "file1.h"
#include "file2.h" // OK
```

El programa *PROG.CPP* incluye dos archivos de encabezado, que contienen `typedef` declaraciones para el nombre `CHAR`. Mientras las declaraciones hagan referencia al mismo tipo, se puede volver a declarar el nombre.

`typedef` No puede volver a definir un nombre que se declaró previamente como un tipo diferente. Por lo tanto, si *archivo2.H* contiene

```
// FILE2.H
typedef int CHAR; // Error
```

el compilador genera un error debido al intento de volver a declarar el nombre `CHAR` para hacer referencia a un tipo diferente. Esto se aplica también a las construcciones como:

```
typedef char CHAR;
typedef CHAR CHAR; // OK: redeclared as same type

typedef union REGS // OK: name REGS redeclared
{
    struct wordregs x; // same meaning.
    struct byteregs h;
} REGS;
```

typedefs en C++ frente a C

El uso del `typedef` especificador con tipos de clase se admite en gran medida debido a la práctica de ANSI C de declarar estructuras sin nombre en `typedef` declaraciones. Por ejemplo, muchos programadores de C usan lo siguiente:

```
// 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:

```
POINT ptOrigin;
```

en lugar de:

```
struct point_t ptOrigin;
```

En C++, la diferencia entre `typedef` los nombres y los tipos reales (declarados con las `class` `struct` `union` palabras clave,, y `enum`) es más distinta. Aunque la práctica de C de declarar una estructura sin nombre en una `typedef` instrucción sigue funcionando, no proporciona ningún beneficio notorio como en 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` con la sintaxis de clase sin nombre `typedef`. `POINT` se trata como un nombre de clase; sin embargo, a los nombres así especificados se aplican las siguientes restricciones:

- El nombre (el sinónimo) no puede aparecer después de un `class` `struct` `union` prefijo, o.
- El nombre no se puede utilizar 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

06/03/2021 • 8 minutes to read • [Edit Online](#)

La `using` declaración 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 nombres de espacio de nombres, clase o enumeración y operadores de resolución de ámbito (::), terminada por un operador de resolución de ámbito. Se puede usar un solo operador de resolución de ámbito para introducir un nombre del espacio de nombres global. La palabra clave `typename` es opcional y se puede usar para resolver nombres dependientes cuando se introducen en una plantilla de clase de una clase base.

inqualified-ID Una expresión de ID incompleta, que puede ser un identificador, un nombre de operador sobre cargado, un nombre de función de conversión o un operador literal definido por el usuario, un nombre de destructor de clase o un nombre de plantilla y una lista de argumentos.

lista de declaradores Lista separada por comas de los `typename` declaradores de *identificador incompleto* del *especificador de nombre anidado* de [], seguido opcionalmente por puntos suspensivos.

Observaciones

Una declaración Using introduce un nombre no completo como sinónimo de una entidad declarada en otro lugar. Permite usar un nombre único de un espacio de nombres específico sin la calificación explícita en la región de declaración en la que aparece. Esto contrasta con la [directiva using](#), que permite usar *todos* los nombres de un espacio de nombres sin calificación. La `using` palabra clave también se usa para los [alias de tipo](#).

Ejemplo: `using` declaración en el campo de clase

Se puede utilizar una declaración using en una definición de clase.

```

// 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');
}

```

```

In D::f()
In B::f()
In B::g()

```

Ejemplo: `using` declaración para declarar un miembro

Cuando se utiliza para declarar un miembro, una declaración `using` debe hacer referencia a un miembro de una clase base.

```

// 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');
}

```

In B::f()

Ejemplo: `using` declaración con calificación explícita

Se puede hacer referencia a los miembros declarados mediante una declaración Using mediante la calificación explícita. El prefijo `::` hace referencia al espacio de nombres global.

```

// 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();
}

```

```

In h
In f
In A::g

```

Ejemplo: `using` sinónimos y alias de declaración

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 `using` declaración es un alias para su nombre original. No afecta al tipo, la vinculación u otros atributos de la declaración original.

```

// 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 y declaraciones locales `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.

```

// 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 y declaraciones de funciones locales `using`

Una declaración de función local no puede tener el mismo nombre y tipo que una función introducida mediante una declaración `using`. Por ejemplo:

```
// 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: `using` declaración 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.

```

// 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);
}

```

```

In D::f(int)
In B::f(char)
In B::g
In D::g(char)

```

Ejemplo: `using` accesibilidad a la declaración

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.

Para obtener más información sobre la accesibilidad de los miembros, vea [member-access control](#).

```
// 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
};
```

Consulta también

[Espacios de nombres](#)

[Palabras clave](#)

volatile (C++)

06/03/2021 • 6 minutes to read • [Edit Online](#)

Calificador de tipo que puede utilizar para declarar que el hardware puede modificar un objeto en el programa.

Sintaxis

```
volatile declarator ;
```

Observaciones

Puede usar el modificador de compilador `/volatile` para modificar el modo en que el compilador interpreta esta palabra clave.

Visual Studio interpreta la `volatile` palabra clave de manera diferente en función de la arquitectura de destino. En el caso de ARM, si no se especifica ninguna opción del compilador `/volatile`, el compilador funciona como si se hubiera especificado `/volatile: ISO`. En el caso de las arquitecturas que no sean ARM, si no se especifica ninguna opción del compilador `/volatile`, el compilador funciona como si se hubiera especificado `/volatile: MS`; por lo tanto, para las arquitecturas que no sean ARM, se recomienda encarecidamente especificar `/volatile: ISO` y usar primitivas de sincronización explícitas y intrínsecos del compilador cuando se trabaja con memoria compartida entre subprocesos.

Puede usar el `volatile` calificador para proporcionar acceso a las ubicaciones de memoria que usan los procesos asincrónicos, como los controladores de interrupción.

Cuando `volatile` se utiliza en una variable que también tiene la palabra clave `_restrict`, `volatile` tiene prioridad.

Si un `struct` miembro está marcado como `volatile`, `volatile` se propaga a la estructura completa. Si una estructura no tiene una longitud que se pueda copiar en la arquitectura actual mediante una instrucción, `volatile` se puede perder por completo en esa estructura.

La `volatile` palabra clave puede no tener ningún efecto en un campo si se cumple alguna de las siguientes condiciones:

- La longitud del campo volátil supera el tamaño máximo que se puede copiar en la arquitectura actual mediante una instrucción.
- La longitud del extremo que contiene `struct`, o si es un miembro de un posiblemente anidado `struct`, supera el tamaño máximo que se puede copiar en la arquitectura actual mediante una instrucción.

Aunque el procesador no reordena los accesos de memoria no almacenables en caché, las variables no almacenables en caché deben marcarse como `volatile` para garantizar que el compilador no reordene los accesos de memoria.

Los objetos que se declaran 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.

Conforme a ISO

Si está familiarizado con la palabra clave `volatile` de C# o está familiarizado con el comportamiento de `volatile` en versiones anteriores del compilador de Microsoft C++ (MSVC), tenga en cuenta que la palabra clave estándar ISO de C++ 11 `volatile` es diferente y se admite en MSVC cuando se especifica la opción del compilador `/volatile: ISO`. (Para ARM, se especifica de forma predeterminada). La `volatile` palabra clave en el código estándar ISO de C++ 11 solo se usará para el acceso al hardware; no la use para la comunicación entre subprocesos. Para la comunicación entre subprocesos, use mecanismos como `STD:: <T> Atomic` desde la biblioteca estándar de C++.

Fin de Conforme a ISO

Específicos de Microsoft

Cuando se usa la opción del compilador `/volatile: MS` (de forma predeterminada, cuando se destina a arquitecturas distintas de ARM), el compilador genera código adicional para mantener el orden entre las referencias a objetos volátiles, además de mantener 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.

NOTE

Cuando se basa en la garantía mejorada que se proporciona cuando se usa la opción del compilador `/volatile: MS`, el código no es portátil.

FIN de Específicos de Microsoft

Vea también

[Palabras clave](#)

[const](#)

[Punteros const y volatile](#)

decltype (C++)

06/03/2021 • 11 minutes to read • [Edit Online](#)

El `decltype` especificador de tipo produce el tipo de una expresión especificada. El `decltype` especificador de tipo, junto con la `auto` palabra clave, es útil principalmente para los desarrolladores que escriben bibliotecas de plantillas. Utilice `auto` y `decltype` para declarar una función de plantilla cuyo tipo de valor devuelto depende de los tipos de sus argumentos de plantilla. O bien, use `auto` y `decltype` para declarar una función de plantilla que contenga una llamada a otra función y, a continuación, devuelva el tipo de valor devuelto de la función ajustada.

Sintaxis

```
* decltype( ***expresión de )
```

Parámetros

Expresiones

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

Valor devuelto

Tipo del parámetro de *expresión*.

Observaciones

El `decltype` especificador de tipo 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 parámetro *Expression*.

- Si el parámetro *Expression* es un identificador o un [acceso de miembro de clase](#), `decltype(expression)` es el tipo de la entidad denominada by *Expression*. Si no hay tal entidad o el parámetro de *expresión* nombra un conjunto de funciones sobrecargadas, el compilador produce un mensaje de error.
- Si el parámetro *Expression* es una llamada a una función o 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 parámetro de *expresión* es un valor `r`, `decltype(expression)` es el tipo de *expresión*. Si el parámetro *Expression* es un valor `l`, `decltype(expression)` es una [referencia lvalue](#) al tipo de *Expression*.

En el ejemplo de código siguiente se muestran algunos usos del `decltype` especificador de tipo. En primer lugar, suponga que ha incluido las siguientes instrucciones en el código.

```
int var;
const int&& fx();
struct A { double x; }
const A* a = new A();
```

A continuación, examine los tipos devueltos por las cuatro `decltype` instrucciones de la tabla siguiente.

	TIPO	NOTAS
<code>decltype(fx());</code>	<code>const int&&</code>	Referencia a un valor <code>r</code> a un <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 dado <code>a</code> que se declara como un <code>const</code> puntero, el tipo es una referencia a <code>const double</code> .

Decltype y Auto

En C++ 14, puede usar sin `decltype(auto)` ningún tipo de valor devuelto final para declarar una función de plantilla 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 función de plantilla cuyo tipo de valor devuelto depende de los tipos de sus argumentos de plantilla. Considere, por ejemplo el ejemplo de código siguiente en el que el tipo de valor devuelto de la función de plantilla depende de los tipos de los argumentos de plantilla. En el ejemplo de código, el marcador de posición *Unknown* indica que no se puede especificar el tipo de valor devuelto.

```
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 función de plantilla. Use la *Sintaxis de declaración de función alternativa* que se muestra más adelante, la `auto` palabra clave y el `decltype` especificador de tipo para declarar un tipo de valor devuelto *especificado en tiempo de ejecución*. El tipo de valor devuelto especificado en tiempo de compilación 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. Tenga en cuenta que los `const` `volatile` calificadores y la `throw` especificación de excepción son opcionales. El marcador de posición `function_body` representa una instrucción compuesta que especifica lo que hace la función. Como práctica recomendada de codificación, el marcador de posición de `expresión` en la `decltype` instrucción debe coincidir con la expresión especificada por la `return` instrucción, si existe, en el `function_body`.

```
auto ***function_name* ( parámetrosOPT OPC ) const volatile -> decltype( expresión opt )
noexcept { *function_body*** };
```

En el ejemplo de código siguiente, el tipo de valor devuelto especificado en tiempo de compilación de la función de plantilla `myFunc` viene determinado por los tipos de los argumentos de plantilla `t` y `u`. Como práctica recomendada de codificación, en el ejemplo de código también se utilizan referencias rvalue y la `forward` plantilla de función, que admiten el *reenvío directo*. Para más información, vea [Declarador de referencia a un valor R: &&](#).

```
//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); }
```

Funciones de reenvío y decltype (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 `decltype` especificador de tipo. El `decltype` especificador de tipo habilita las funciones de reenvío genéricas porque no pierde información necesaria sobre si una función devuelve un tipo de referencia. Para obtener un ejemplo de código de una función de reenvío, vea el ejemplo anterior de la función de plantilla `myFunc`.

Ejemplos

En el ejemplo de código siguiente se declara el tipo de valor devuelto especificado en tiempo de compilación de la función de plantilla `Pplus()`. La `Pplus` función procesa sus dos operandos con la `operator+` sobrecarga. Por consiguiente, la interpretación del operador más (`+`) y el tipo de valor devuelto de la `Pplus` función depende de los tipos de los argumentos de la función.

```

// 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;
}

```

```

Plus(i, 9) = 13
Plus(dx, dy) = 13.5
Hello, world!
x3.Dump() = 42

```

Visual Studio 2017 y versiones posteriores: El compilador analiza `decltype` los argumentos cuando se declaran las plantillas en lugar de crear una instancia. Por lo tanto, si se encuentra una especialización no dependiente en el `decltype` argumento, no se diferirá en el momento de la creación de instancias y se procesará inmediatamente y los errores resultantes se diagnosticarán en ese momento.

En el ejemplo siguiente se muestra este tipo de error del compilador que se genera en el punto de declaración:

```
#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 posterior.

Atributos en C++

06/03/2021 • 8 minutes to read • [Edit Online](#)

El estándar de C++ define un conjunto de atributos y también permite a los proveedores de compiladores definir sus propios atributos (dentro de un espacio de nombres específico del proveedor), pero los compiladores son necesarios para reconocer solo los atributos definidos en el estándar.

En algunos casos, los atributos estándar se superponen con los parámetros `declspec` específicos del compilador. En Visual C++, puede usar el `[[deprecated]]` atributo en lugar de usar `declspec(deprecated)` y el atributo será reconocido por cualquier compilador compatible. En el caso de todos los demás parámetros `declspec` como `dllimport` y `dllexport`, todavía 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 posterior (disponible con [/STD: c++ 17](#)): en el ámbito de una lista de atributos, puede especificar el espacio de nombres para todos los nombres con un solo `using` presentador:

```
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 que puede ser o no específica del proveedor. Un compilador puede utilizar esta información para generar mensajes informativos o para aplicar lógica especial al compilar el código con atributos. El compilador omite cualquier atributo que no reconozca, lo que significa que no puede definir sus propios atributos personalizados mediante esta sintaxis. Los atributos se incluyen entre corchetes dobles:

```
[[deprecated]]
void Foo(int);
```

Los atributos representan una alternativa normalizada a las extensiones específicas del proveedor, como las directivas de `#pragma`, `__declspec ()` (Visual C++) o `__attribute__` de atributo (GNU). Sin embargo, todavía necesitará usar las construcciones específicas del proveedor para la mayoría de los propósitos. El estándar especifica actualmente los siguientes atributos que debe reconocer un compilador compatible:

- `[[noreturn]]` Especifica que una función no devuelve nunca; en otras palabras, siempre produce una excepción. El compilador puede ajustar sus reglas de compilación para las `[[noreturn]]` entidades.
- `[[carries_dependency]]` Especifica que la función propaga la ordenación de las dependencias de datos con respecto a la sincronización de subprocessos. El atributo se puede aplicar a uno o varios parámetros, para especificar que el argumento pasado incluye una dependencia en el cuerpo de la función. El atributo se puede aplicar a la propia función, para especificar que el valor devuelto incluye una dependencia de la función. El compilador puede utilizar esta información para generar código más eficaz.
- `[[deprecated]]` **Visual Studio 2015 y versiones posteriores:** Especifica que una función no está diseñada para utilizarse y podría no existir en versiones futuras de una interfaz de biblioteca. El

compilador puede utilizar esto para generar un mensaje informativo cuando el código de cliente intenta llamar a la función. Se puede aplicar a la declaración de una clase, un nombre de TypeDef, una variable, un miembro de datos no estático, una función, un espacio de nombres, una enumeración, un enumerador o una especialización de plantilla.

- **[[fallthrough]] Visual Studio 2017 y versiones posteriores:** (disponible con [/STD: c++ 17](#)) el **[[fallthrough]]** atributo se puede usar en el contexto de las instrucciones [Switch](#) como una sugerencia para el compilador (o cualquier persona que lea el código) para el que está previsto el comportamiento de fallthrough. Actualmente, el compilador de Microsoft C++ no advierte sobre el comportamiento de fallthrough, por lo que este atributo no tiene ningún efecto en el comportamiento del compilador.
- **[[nodiscard]] Visual Studio 2017 versión 15,3 y versiones posteriores:** (disponible con [/STD: c++ 17](#)) especifica que el valor devuelto de una función no está diseñado para descartarse. Genera una advertencia C4834, como se muestra en este ejemplo:

```
[[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](#)) especifica que no se puede usar intencionadamente una variable, función, clase, TypeDef, miembro de datos no estático, enumeración o especialización de plantilla. El compilador no advierte cuando **[[maybe_unused]]** no se usa una entidad marcada. Una entidad que se declara sin el atributo se puede volver a declarar más adelante con el atributo y viceversa. Una entidad se considera marcada después de la primera declaración que está marcada como analizada y para el resto de la traducción de la unidad de traducción actual.

Atributos específicos de Microsoft

- **[[gsl::suppress(rules)]]** Este atributo específico de Microsoft se usa para suprimir las advertencias de los comprobadores que aplican las reglas de la [biblioteca de compatibilidad de directrices \(GSL\)](#) en el código. Por ejemplo, considere este fragmento de código:

```
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:

- 26494 (regla de tipo 5: inicializar siempre un objeto).
- 26485 (regla de límites 3: no se decadencia de matriz a puntero).
- 26481 (regla de límites 1: no usar aritmética de punteros. En su lugar, use span).

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 el perfil de límite completo escribiendo [[GSL:: Suppress (Bounds)]] sin incluir un número de regla específico. Los C++ Core Guidelines están diseñados para ayudarle a escribir código mejor y más seguro. El atributo Suppress facilita la desactivación de las advertencias cuando no se desean.

Operadores integrados de C++, prioridad y asociatividad

06/03/2021 • 4 minutes to read • [Edit Online](#)

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 *precedencia* de operadores especifica el orden de las operaciones en expresiones que contienen más de un operador. La *asociatividad* del operador especifica si, en una expresión que contiene varios operadores con la misma prioridad, un operando se agrupa con el que se encuentra a la izquierda o a la derecha.

Ortografías alternativas

C++ especifica las correcciones alternativas para algunos operadores. En C, las correcciones alternativas se proporcionan como macros en el <iso646.h> encabezado. En C++, estas alternativas son palabras clave y el uso de <iso646.h> o el equivalente de C++ <ciso646> está en desuso. En Microsoft C++, `/permissive-` `/Za` se requiere la opción del compilador o para habilitar las correcciones alternativas.

Prioridad de los operadores de C++ y tabla de asociatividad

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	OPERATOR	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>. de -></code>	
Subíndice de matriz	<code>[]</code>	
Llamada a función	<code>()</code>	
Incremento de postfijo	<code>++</code>	
Decremento de postfijo	<code>--</code>	

DESCRIPCIÓN DEL OPERADOR	OPERATOR	ALTERNATIVA
Nombre de tipo	<code>typeid</code>	
Conversión de tipos constante	<code>const_cast</code>	
Conversión de tipos dinámica	<code>dynamic_cast</code>	
Conversión de tipos reinterpretada	<code>reinterpret_cast</code>	
Conversión de tipos estática	<code>static_cast</code>	
Precedencia de grupo 3, asociatividad de derecha a izquierda		
Tamaño de objeto o tipo	<code>sizeof</code>	
Incremento de prefijo	<code>++</code>	
Decremento de prefijo	<code>--</code>	
Complemento de uno	<code>~</code>	<code>compl</code>
Not lógico	<code>!</code>	<code>not</code>
Negación unaria	<code>-</code>	
Suma unaria	<code>+</code>	
Dirección de	<code>&</code>	
Direccionamiento indirecto	<code>*</code>	
Crear objeto	<code>new</code>	
Destruir objeto	<code>delete</code>	
Conversión	<code>()</code>	
Precedencia de grupo 4, asociatividad de izquierda a derecha		
Puntero a miembro (objetos o punteros)	<code>.* de ->*</code>	
Precedencia de grupo 5, asociatividad de izquierda a derecha		
Multiplicación	<code>*</code>	

DESCRIPCIÓN DEL OPERADOR	OPERATOR	ALTERNATIVA
Ella	/	
Módulo	%	
Precedencia de grupo 6, asociatividad de izquierda a derecha		
Agregado	+	
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 de izquierda a derecha		
AND bit a bit	&	bitand
Precedencia de grupo 11, asociatividad de izquierda a derecha		
OR exclusivo bit a bit	^	xor

DESCRIPCIÓN DEL OPERADOR	OPERATOR	ALTERNATIVA
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
Precedencia de grupo 15, asociatividad de derecha a izquierda		
Condicional	? :	
Asignación	=	
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	

DESCRIPCIÓN DEL OPERADOR	OPERATOR	ALTERNATIVA
Precedencia de grupo 16, asociatividad de izquierda a derecha		
Coma	,	

Consulte también

[Sobrecarga de operadores](#)

alignof (operador)

06/03/2021 • 2 minutes to read • [Edit Online](#)

El `alignof` operador devuelve la alineación en bytes del tipo especificado como un valor de tipo `size_t`.

Sintaxis

```
alignof( type )
```

Observaciones

Por ejemplo:

EXPRESSION	VALOR
<code>alignof(char)</code>	1
<code>alignof(short)</code>	2
<code>alignof(int)</code>	4
<code>alignof(long long)</code>	8
<code>alignof(float)</code>	4
<code>alignof(double)</code>	8

El `alignof` valor es el mismo que el valor de `sizeof` para los tipos básicos. Considere, no obstante, este ejemplo:

```
typedef struct { int a; double b; } S;
// alignof(S) == 8
```

En este caso, el `alignof` valor es el requisito de alineación del elemento más grande de la estructura.

De igual forma, para

```
typedef __declspec(align(32)) struct { int a; } S;
```

`alignof(S)` es igual a `32`.

Un uso de `alignof` sería como parámetro de 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.

```
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 \(alineación de miembros de estructura\)](#)
- [Ejemplos de alineación de estructura](#) (específico de 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 el estándar de C++ 11, el operador específico de Microsoft `_alignof` proporcionaba esta funcionalidad. Para obtener la máxima portabilidad, debe usar el `alignof` operador en lugar del operador específico de Microsoft `_alignof`.

Por 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`

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Recupera el GUID asociado a la expresión.

Sintaxis

```
* _uuidof ( ***expresión de )
```

Observaciones

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 formado por ceros.

Utilice esta palabra clave para extraer el GUID asociado a lo siguiente:

- Objeto por el `uuid` atributo extendido.
- Bloque de biblioteca creado con el `module` atributo.

NOTE

En una compilación de depuración, `_uuidof` siempre Inicializa un objeto dinámicamente (en tiempo de ejecución). En una versión de lanzamiento, `_uuidof` puede inicializar estáticamente (en tiempo de compilación) un objeto.

Por 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:

```
// 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 lpoolestr;
    StringFromCLSID(__uuidof(MyLib), &lpoolestr);
    wprintf_s(L"%s", lpoolestr);
    CoTaskMemFree(lpoolestr);
}
```

Comentarios

En los casos en los que el nombre de la biblioteca ya no está en el ámbito, puede usar en `__LIBID_` lugar de `__uuidof`. Por ejemplo:

```
StringFromCLSID(__LIBID_, &lpoolestr);
```

FIN de Específicos de Microsoft

Vea también

[Expresiones con operadores unarios](#)

[Palabras clave](#)

Operadores de adición: + y -

06/03/2021 • 5 minutes to read • [Edit Online](#)

Sintaxis

```
expression + expression  
expression - expression
```

Observaciones

Los operadores aditivos son:

- Suma (+)
- 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>íntegra</i>	Los tipos char e int de todos los tamaños (long, short) y las enumeraciones son tipos "enteros".
<i>escalar</i>	Los operandos escalares son operandos de tipo aritmético o puntero.

Las combinaciones válidas para estos operadores son:

operaciones aritméticas + operaciones aritméticas

escalar + entero de

entero + de escalar

operaciones aritméticas - operaciones aritméticas

escalar - escalar

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

operando y el resultado es del tipo convertido.

Ejemplo

```
// 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:

```
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)`).

NOTE

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 matriz ($n - i$) TH, donde n es el elemento al que apunta el puntero original e i es el valor entero del segundo operando.

Consulta también

Expresiones con operadores binarios

Operadores integrados de C++, precedencia y asociatividad

Operadores de adición de C

Operador Address-of: &

02/11/2020 • 4 minutes to read • [Edit Online](#)

Syntax

& *cast-expression*

Comentarios

El operador unario Address-of (`&`) toma la dirección de su operando. El operando del operador Address-of puede ser un designador de función o un valor *I* que designe un objeto que no sea un campo de bits.

El operador Address-of solo se puede aplicar a variables de tipos fundamentales, de estructura, de clase o de Unión que se declaran en el nivel de ámbito de archivo o a referencias de matriz de subíndice. En estas expresiones, se puede Agregar o restar a la expresión de dirección una expresión constante que no incluya el operador Address-of.

Cuando se aplica a funciones o valores *L*, el resultado de la expresión es un tipo de puntero (un valor *R*) 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, que se aplica a los `const` `volatile` objetos o, `const type *` se evalúa como `volatile type *`, donde `type` es el tipo del objeto original.

La dirección de una función sobrecargada solo se puede realizar cuando está claro a qué versión de la función se hace referencia. Vea [sobrecarga de funciones](#) para obtener información sobre cómo obtener la dirección de una función sobrecargada determinada.

Cuando el operador Address-of se aplica a un nombre completo, el resultado depende de si el *nombre completo* especifica un miembro estático. En ese caso, el resultado es un puntero al tipo especificado en la declaración del miembro. Para un miembro que no es estático, el resultado es un puntero al *nombre* de miembro de la clase indicada por *el nombre de clase calificado*. Para obtener más información sobre *Qualified-Class-Name*, vea [expresiones primarias](#).

Ejemplo: dirección del miembro estático

En el fragmento de código siguiente se muestra cómo difiere el resultado del operador Address-of, dependiendo de si un miembro de clase es estático:

```
// 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:

```
// 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;
}
```

```
&d equals &rd
```

Ejemplo: dirección de la función como parámetro

En el ejemplo siguiente se usa el operador address-of para pasar un argumento de puntero a una función:

```
// 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
}
```

25

Vea también

[Expresiones con operadores unarios](#)

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

[Declarador de referencia a un valor L: &](#)

[Operadores de direccionamiento indirecto y address-of](#)

Operadores de asignación

02/11/2020 • 11 minutes to read • [Edit Online](#)

Sintaxis

expression asignación de expresión: expresión de operador expression

assignment-operator: uno de

= *= /= %= += -= <<= >>= &= ^= |=

Observaciones

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, excepto 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.
-=	Resta el valor del segundo operando del valor del primer operando; almacena el resultado en el objeto especificado por el primer operando.

OPERATOR	SIGNIFICADO
<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:

OPERATOR	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 ortografías alternativas para los operadores de asignación compuesta. En C, las correcciones alternativas se proporcionan como macros en el `<iso646.h>` encabezado. En C++, las correcciones alternativas son palabras clave; el uso de `<iso646.h>` o el equivalente de C++ `<ciso646>` está en desuso. En Microsoft C++, `/permissive-` `/Za` se requiere la opción del compilador o para habilitar la ortografía alternativa.

Ejemplo

```

// 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 `const` de `volatile` tipos y 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` tipos, 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 de *movimiento*. Para obtener más información, vea [constructores de copia y operadores de asignación de copia](#) y [constructores de movimiento y operadores de asignación de movimiento](#).

Un objeto de cualquier clase derivada sin ambigüedad de una clase base se puede asignar a un objeto de la clase base. Inverso 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

```

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

```

UserType1 A;
UserType2 B;

B = A;

```

la instrucción de asignación

```
B = A;
```

puede tener uno de los efectos siguientes:

- Llame a la función `operator=` para `UserType2`, `operator=` se proporciona un `UserType1` argumento.
- 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 $E1 \text{ OP} = E2$, donde $E1$ es un `const` valor l no modifiable y $E2$ es:

- un tipo aritmético
- un puntero, si OP es `+` o`**` `-` `**`

El formulario $E1 \text{ OP} = E2$ se comporta como $E1 = E1 \text{ OP} E2$, pero $E1$ solo se evalúa una vez.

La asignación compuesta a un tipo enumerado genera un mensaje de error. Si el operando izquierdo es de un tipo de puntero, el operando derecho debe ser de un tipo de puntero o debe ser una expresión constante que se evalúe como 0. Cuando el operando izquierdo es de un tipo entero, el operando derecho no debe ser de un tipo de puntero.

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 legal `(a += b) += c` de C++ no se permite en C.

Consulte también

[Expresiones con operadores binarios](#)

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

[Operadores de asignación de C](#)

Operador and bit a bit:&

02/11/2020 • 2 minutes to read • [Edit Online](#)

Sintaxis

`expresión & de expresión de`

Observaciones

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 del 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 `<iso646.h>` encabezado. 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++, `/permissive-` `/za` se requiere la opción del compilador o para habilitar la ortografía alternativa.

Ejemplo

```
// expte_Bitwise_AND_Operator.cpp
// compile with: /EHsc
// Demonstrate bitwise AND
#include <iostream>
using namespace std;
int main() {
    unsigned short a = 0xFFFF;      // pattern 1111 ...
    unsigned short b = 0xAAAA;      // pattern 1010 ...

    cout << hex << ( a & b ) << endl;   // prints "aaaa", pattern 1010 ...
}
```

Consulte también

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

[Operadores bit a bit de C](#)

Operador OR exclusivo bit a bit: ^

02/11/2020 • 2 minutes to read • [Edit Online](#)

Syntax

`expresión ^ de expresión de`

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 de resultado correspondiente se establece en 1. De lo contrario, el bit del resultado correspondiente se establece en 0.

Ambos operandos del operador deben tener tipos enteros. Las conversiones aritméticas habituales descritas en [conversiones estándar](#) se aplican a los operandos.

Para obtener más información sobre el uso alternativo del `^` carácter en C++/CLI y C++/CX, vea [identificador del operador de objeto \(^\) \(C++/CLI y C++/CX\)](#).

Operator (palabra clave) para ^

C++ especifica `xor` como una ortografía alternativa para `^`. En C, la ortografía alternativa se proporciona como una macro en el `<iso646.h>` encabezado. 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++, `/permissive-` `/Za` se requiere la opción del compilador o para habilitar la ortografía alternativa.

Ejemplo

```
// 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 ...
}
```

Vea también

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

Operador OR inclusivo bit a bit: |

02/11/2020 • 2 minutes to read • [Edit Online](#)

Sintaxis

`expression1 | expression2`

Observaciones

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 del operador deben tener tipos enteros. Las conversiones aritméticas habituales descritas en [conversiones estándar](#) 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 `<iso646.h>` encabezado. 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++, `/permissive-` `/za` se requiere la opción del compilador o para habilitar la ortografía alternativa.

Ejemplo

```
// expe_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++, prioridad y asociatividad](#)

[Operadores bit a bit de C](#)

Operador de conversión: ()

06/03/2021 • 2 minutes to read • [Edit Online](#)

Una conversión de tipo proporciona un método para la conversión explícita del tipo de un objeto en una situación concreta.

Sintaxis

```
unary-expression ( type-name ) cast-expression
```

Observaciones

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. Se pueden aplicar otras restricciones derivadas de los tamaños reales o de la representación de tipos específicos.

Ejemplos

```
// 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;
}
```

```

// 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 = (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);
}

```

```

RawBytes truncated to 10 chars:  Exciting!!
Casted Bytes:  Exciting
Note that the cast changed the raw internal string
Raw Bytes after cast:  Exciting

```

Vea también

[Expresiones con operadores unarios](#)

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

[Operador de conversión explícita de tipos: \(\)](#)

[Operadores de conversión](#)

[Operadores de conversión](#)

Operador coma: ,

06/03/2021 • 2 minutes to read • [Edit Online](#)

Permite agrupar dos instrucciones donde se espera una.

Sintaxis

```
expression , expression
```

Observaciones

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.

Consideré la expresión `e1, e2`. El tipo y el valor de la expresión son el tipo y el valor de *E2*; se descarta el resultado de la evaluación de *E1*. 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:

```
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

```
// 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);
}
```

Consulte también

[Expresiones con operadores binarios](#)

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

[Operador de evaluación secuencial](#)

Operador condicional: ?: :

06/03/2021 • 4 minutes to read • [Edit Online](#)

Sintaxis

```
expression ? expression : expression
```

Observaciones

El operador condicional (?:) es un operador ternario (toma tres operandos). El operador condicional funciona del modo siguiente:

- El primer operando se convierte implícitamente en `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 aritméticos 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 l.

WARNING

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

```
// 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++)

06/03/2021 • 5 minutes to read • [Edit Online](#)

Desasigna un bloque de memoria.

Sintaxis

```
[ :: ] delete Cast: expresión  
[ :: ] delete [] Cast: expresión
```

Observaciones

El argumento *Cast-Expression* debe ser un puntero a un bloque de memoria previamente asignado para un objeto creado con el [operador New](#). El `delete` operador tiene un resultado de tipo `void` y, por tanto, no devuelve un valor. Por ejemplo:

```
CDialog* MyDialog = new CDialog;  
// use MyDialog  
delete MyDialog;
```

`delete` El uso de en un puntero a un objeto no asignado con `new` proporciona resultados imprevisibles. Sin embargo, puede usar `delete` en un puntero con el valor 0. Esta disposición significa que, cuando `new` devuelve 0 en caso de error, es inofensivo eliminar el resultado de una operación con errores `new`. Para obtener más información, vea [los operadores New y DELETE](#).

Los `new` `delete` operadores y también se pueden usar para los tipos integrados, incluidas las matrices. Si `pointer` hace referencia a una matriz, coloque corchetes vacíos (`[]`) antes de `pointer`:

```
int* set = new int[100];  
//use set[]  
delete [] set;
```

El uso del `delete` operador en un objeto desasigna su memoria. Un programa que desreferencia un puntero después de eliminarse el objeto puede producir resultados impredecibles o bloquearse.

Cuando `delete` se usa para desasignar memoria para un objeto de clase de C++, se llama al destructor del objeto antes de que se cancele la asignación de la memoria del objeto (si el objeto tiene un destructor).

Si el operando del `delete` operador es un valor l modifiable, su valor queda sin definir después de eliminar el objeto.

Si se especifica la opción del compilador [/SDL \(habilitar comprobaciones de seguridad adicionales\)](#), el operando del `delete` operador 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 individuales y la otra para matrices de objetos. En el fragmento de código siguiente se muestra cómo difieren:

```

// 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 producen resultados sin definir: usar la forma de matriz de Delete (`delete []`) en un objeto y usar la forma de no matriz de DELETE en una matriz.

Ejemplo

Para obtener ejemplos del uso de `delete`, vea [New \(operador\)](#).

Cómo eliminar trabajos

El operador Delete invoca la función de **operador Delete**.

Para los objetos que no son de tipo de clase ([Class](#), [structo](#) [Union](#)), 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.

Consulta también

[Expresiones con operadores unarios](#)

[Palabra](#)

[Operadores New y DELETE](#)

Operadores de igualdad: == y !=

02/11/2020 • 2 minutes to read • [Edit Online](#)

Sintaxis

```
expresión == de expresión de
expresión != de expresión de
```

Observaciones

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 menor prioridad que los operadores relacionales, pero se comportan de manera similar. 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 not-Equal-to (`!=`) 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 `<iiso646.h>` encabezado. En C++, la ortografía alternativa es una palabra clave; el uso de `<iiso646.h>` o el equivalente de C++ `<ciso646>` está en desuso. En Microsoft C++, `/permissive-` `/Za` se requiere la opción del compilador o para habilitar la ortografía alternativa.

Ejemplo

```
// 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 este tipo de 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++, prioridad; y asociatividad](#)

Operadores relacionales y de igualdad de C

Operador de conversión explícita de tipos: ()

06/03/2021 • 3 minutes to read • [Edit Online](#)

C++ permite la conversión de tipos explícita mediante una sintaxis similar a la de las llamadas de función.

Sintaxis

```
simple-type-name ( expression-list )
```

Observaciones

Un *simple-type-name* seguido de una *expresión-List* entre paréntesis crea un objeto del tipo especificado mediante las expresiones especificadas. En el ejemplo siguiente se muestra una conversión de tipo explícita al tipo int:

```
int i = int( d );
```

En el ejemplo siguiente se muestra una `Point` clase.

Ejemplo

```

// expte_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();
}

```

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:

```

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:

```
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:

```
struct Point
{
    Point( short x, short y ) { _x = x; _y = y; }
    ...
    short _x, _y;
};

...
Point pt = Point( 3, 10 );
```

En el ejemplo anterior, que utiliza la conversión de estilo de función, se muestra cómo convertir dos valores (uno para `x` y otro para `y`) al tipo definido por el usuario `Point`.

Caution

Utilice las conversiones de tipos explícitas con cuidado, ya que invalidan la comprobación de tipos integrada del compilador de C++.

La notación de [conversión](#) se debe usar para las conversiones a tipos que no tienen un *nombre de tipo simple* (por ejemplo, un puntero o un tipo de referencia). La conversión a tipos que se pueden expresar con un *simple-type-name* se puede escribir en cualquier forma.

La definición de tipos en las conversiones "cast" no es válida.

Consulta también

[Expresiones de postfijo](#)

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

Operador de llamada de función: ()

06/03/2021 • 6 minutes to read • [Edit Online](#)

Una llamada de función es un tipo de `postfix-expression`, formado por una expresión que se evalúa como una función o un objeto al que se puede llamar seguido del operador de llamada a función, `()`. Un objeto puede declarar una `operator ()` función, que proporciona la semántica de llamada de función para el objeto.

Sintaxis

```
postfix-expression :  
    postfix-expression ( argument-expression-list opt )
```

Observaciones

Los argumentos para el operador de llamada a función proceden de una `argument-expression-list` lista de expresiones separadas por comas. Los valores de estas expresiones se pasan a la función como argumentos. `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 especifica y puede producirse en cualquier orden. En C++ 17 y versiones posteriores, la expresión de función se evalúa antes que cualquier expresión de argumento o argumentos predeterminados. Las expresiones de argumento se evalúan en una secuencia indeterminada.

Se `postfix-expression` evalúa como la función a la que se va a llamar. Puede adoptar cualquiera de las formas siguientes:

- identificador de función, visible en el ámbito actual o en el ámbito de cualquiera de los argumentos de función proporcionados.
- expresión que se evalúa como una función, un puntero a función, un objeto al que se puede llamar o una referencia a una.
- 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 sobrecarga determinan la función real a la que se va a llamar. Si la función miembro es virtual, la función a la que se llama se determina en tiempo de ejecución.

Algunas declaraciones de ejemplo:

- Una función que devuelve el tipo `T`. Una declaración de ejemplo es

```
T func( int i );
```

- Un puntero a una función que devuelve el tipo `T`. Una declaración de ejemplo es

```
T (*func)( int i );
```

- Una referencia a una función que devuelve el tipo `T`. Una declaración de ejemplo es

```
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:

```
(pObject->*pmf)();  
(Object.*pmf)();
```

Ejemplo

En el ejemplo siguiente se llama a la función de biblioteca estándar `strcat_s` con tres argumentos:

```
// 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;
}
```

```
Welcome to C++
```

Resultados de la llamada de función

Una llamada de función se evalúa como un valor `r` a menos que la función se declare como un tipo de referencia. Las funciones con tipos de valor devuelto de referencia se evalúan como `lvalues`. Estas funciones se pueden usar en el lado izquierdo de una instrucción de asignación, como se muestra aquí:

```

// 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";
}

```

En el código anterior se define una clase denominada `Point`, que contiene objetos de datos privados que representan las 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 válido:

```
// 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 acerca de las declaraciones de función, vea [funciones](#). El material relacionado está en [unidades de traducción y vinculación](#).

Consulta también

[Expresiones de postfijo](#)

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

[Llamada a función](#)

Operador de direccionamiento indirecto: *

06/03/2021 • 2 minutes to read • [Edit Online](#)

Sintaxis

```
* cast-expression
```

Observaciones

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 una 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:

```
// 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](#)

address-of (Operador): &

Operadores de direccionamiento indirecto y dirección de

Operadores de desplazamiento a la izquierda y a la derecha (>> y <<)

06/03/2021 • 9 minutes to read • [Edit Online](#)

Los operadores de desplazamiento bit a bit son el operador de desplazamiento `>>` a la derecha (), que mueve los bits de la *expresión Mayús* a la derecha y el operador de desplazamiento a la izquierda (`<<`), que mueve los bits de la *expresión Mayús* a la izquierda.¹

Sintaxis

```
Shift-Expression << aditivo-expresión  
shift-expression >> additive-expression
```

Observaciones

IMPORTANT

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 de desplazamiento a la derecha es significativamente diferente en Windows para dispositivos ARM. Para obtener más información, vea la sección "operadores de desplazamiento" de 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 a la izquierda el número de posiciones especificado por *Additive-Expression*. Las posiciones de bits que quedan vacantes debido a la operación de desplazamiento se llenan 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 [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](#).

```

#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; // 0b00000000'0000100

    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; // 0b00000000'000010000
}

```

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 bit 1 se desplaza a la izquierda en la posición de bit de signo.

```

#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 a 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 llenan 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.

IMPORTANT

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:

```

#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.

```

#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.

```

#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 descritas en [conversiones estándar](#). El tipo del resultado es el mismo que el tipo de la *expresión Shift* promovida.

En el ejemplo siguiente, una variable de tipo `char` se promueve a `int`.

```

#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 adicionales

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 la *expresión Shift*(promovida). No se realiza ninguna operación de desplazamiento si *Additive-Expression* es 0.

```

#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 describe 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 $E1 \ll E2$ 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 del resultado es $E1 \times 2^{E2}$, el módulo menos uno más que el valor máximo que se puede representar en el tipo de 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 de resultado, ese valor, convertido al tipo de resultado, es el valor resultante; de lo contrario, el comportamiento es indefinido.

El valor de $E1 \gg E2$ 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.

Consulta también

[Expresiones con operadores binarios](#)

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

Operador lógico AND:&&

02/11/2020 • 3 minutes to read • [Edit Online](#)

Sintaxis

`expresión && de expresión de`

Observaciones

El operador AND lógico (`&&`) devuelve `true` si ambos operandos son `true` y, de `false` lo contrario, devuelve. 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 AND lógico no necesitan tener el mismo tipo, pero deben tener un tipo booleano, entero o puntero. Los operandos son normalmente expresiones relacionales o de igualdad.

El primer operando se evalúa completamente y todos los efectos secundarios se completan antes de que continúe la evaluación de la expresión AND lógica.

El segundo operando se evalúa solo si el primer operando 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:

```
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 `<iso646.h>` encabezado. 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++, `/permissive-` `/Za` se requiere la opción del compilador o para habilitar la ortografía alternativa.

Ejemplo

```
// 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++, prioridad y asociatividad](#)

[Operadores lógicos de C](#)

Operador de negación lógico: !

02/11/2020 • 2 minutes to read • [Edit Online](#)

Sintaxis

* `!` ***CAST (expresión)

Observaciones

El operador lógico de negación (`!`) invierte el significado de su 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`.

En el caso de una expresión `e`, la expresión unaria `!e` es equivalente a la expresión `(e == 0)`, excepto en los casos en los que intervienen los 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 `<iso646.h>` encabezado. 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++, `/permissive-` `/za` se requiere la opción del compilador o para habilitar la ortografía alternativa.

Ejemplo

```
// 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++, prioridad y asociatividad](#)

[Operadores aritméticos unarios](#)

Operador lógico OR: ||

02/11/2020 • 3 minutes to read • [Edit Online](#)

Sintaxis

`Logical-or-Expression || Logical-and-Expression`

Observaciones

El operador lógico OR (`||`) devuelve el valor booleano `true` si uno o ambos operandos son `true` y, de `false` lo contrario, devuelve. 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 OR lógico no tienen que tener el mismo tipo, pero deben ser de tipo booleano, entero o 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 or lógica es `true`. Se conoce como evaluación *de cortocircuito*.

```
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 se `printf` evalúa como `true`, que se promueve a un 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 evalúa como `true`, la evaluación se detiene.

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 `<iso646.h>` encabezado. 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++, `/permissive-` `/Za` se requiere la opción del compilador o para habilitar la ortografía alternativa.

Ejemplo

```
// 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++, prioridad y asociatividad](#)

[Operadores lógicos de C](#)

Operadores de acceso a miembros:. etc>

06/03/2021 • 2 minutes to read • [Edit Online](#)

Sintaxis

```
postfix-expression . name  
postfix-expression -> name
```

Observaciones

Los operadores de acceso a miembros . y -> se usan para hacer referencia a miembros de estructuras, uniones y clases. 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 el primer formulario, *postfijo-Expression* representa un valor de struct, Class o Union Type, y *Name* nombra un miembro de la estructura, Unión o clase especificadas. El valor de la operación es el de *Name* y es un valor l si *postfijo-Expression* es un valor l.
2. En el segundo formulario, *postfijo-Expression* representa un puntero a una estructura, Unión o clase, y *Name* nombra un miembro de la estructura, Unión o clase especificadas. El valor es el de *Name* y es un valor l. El -> operador desreferencia el puntero. Por consiguiente, las expresiones `e->member` y `(*e).member` (donde *e* representa un puntero) producen resultados idénticos (excepto cuando los operadores -> o * están sobrecargados).

Ejemplo

En el ejemplo siguiente se muestran dos formas del operador de acceso a miembros.

```
// 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;  
}
```

2/1/1900

2/1/2000

Consulta 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

06/03/2021 • 4 minutes to read • [Edit Online](#)

Sintaxis

```
expression * expression
expression / expression
expression % expression
```

Observaciones

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 el sentido de 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 descritas 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 dado 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:

```
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.

NOTE

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.

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 la división integral es -3.

La relación entre los operadores de multiplicación se proporciona mediante la identidad $(E1 / E2) * E2 + E1 \% E2 == E1$.

Ejemplo

El siguiente programa muestra los operadores multiplicativos. Tenga en cuenta que cualquier operando de `10 / 3` debe convertirse explícitamente al tipo `float` para evitar el truncamiento, de modo que ambos operandos sean de tipo antes de la `float` división.

```
// 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;
}
```

Consulta también

[Expresiones con operadores binarios](#)

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

[Operadores de multiplicación de C](#)

new (Operador) (C++)

06/03/2021 • 14 minutes to read • [Edit Online](#)

Asigna memoria para un objeto o matriz de objetos de *tipo-nombre* del almacén libre y devuelve un puntero distinto de cero y con tipo apropiado al objeto.

NOTE

Extensiones de componentes de Microsoft C++ proporciona compatibilidad para `new` que la palabra clave agregue entradas de ranura de vtable. Para obtener más información, vea [New \(nueva ranura en vtable\)](#).

Sintaxis

```
[::] new [placement] new-type-name [new-initializer]  
[::] new [placement] ( type-name ) [new-initializer]
```

Observaciones

Si no se realiza correctamente, `new` devuelve cero o produce una excepción; vea [los operadores New y DELETE](#) para obtener más información. Puede cambiar este comportamiento predeterminado escribiendo una rutina de control de excepciones personalizada y llamando a la función de la biblioteca en tiempo de ejecución de `_set_new_handler` con el nombre de función como su argumento.

Para obtener información sobre cómo crear un objeto en el montón administrado, vea [gcnew](#).

Cuando `new` se utiliza para asignar memoria para un objeto de clase de C++, se llama al constructor del objeto después de asignar la memoria.

Use el operador [Delete](#) para desasignar la memoria asignada con el `new` operador.

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 la matriz del extremo izquierdo puede ser cualquier expresión que se evalúe como un valor positivo. Al asignar una matriz mediante el `new` operador, la primera dimensión puede ser cero; el `new` operador devuelve un puntero único.

```
char (*pchar)[10] = new char[dim][10];  
delete [] pchar;
```

Type-name no puede contener `const`, `volatile`, declaraciones de clase ni declaraciones de enumeración. Por consiguiente, la siguiente expresión no es válida:

```
volatile char *vch = new volatile char[20];
```

El `new` operador no asigna tipos de referencia porque no son objetos.

`new` No se puede usar el operador para asignar una función, pero se puede usar para asignar punteros a funciones. En el ejemplo siguiente se asigna y se libera una matriz de siete punteros a funciones que devuelven enteros.

```
int (**p) () = new (int (*[7]) ());
delete *p;
```

Si usa el operador `new` sin ningún argumento adicional y compila con la opción `/GX`, `/EHA` o `/EHS`, el compilador generará 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`:

Ubicación

Proporciona una manera de pasar argumentos adicionales si se sobrecarga `new`.

nombre de tipo

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.

initializer

Proporciona un valor para el objeto inicializado. No se pueden especificar inicializadores para matrices. El operador `new` creará matrices de objetos solo si la clase tiene un constructor predeterminado.

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.

```

// 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;
}

```

Ejemplo: `new` operador

Si utiliza el nuevo formulario de colocación del `new` operador, el formulario con argumentos además del tamaño de la asignación, el compilador no admite un formulario de colocación del `delete` operador si el constructor produce una excepción. Por ejemplo:

```

// 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 cannot occur.

        A* pa2 = new(__FILE__, __LINE__) A(20);
    } catch (...) {
    }
}

int main() {
    A a;
}

```

Inicializar objetos asignados con new

En la gramática del operador se incluye un campo de *inicializador* opcional `new`. Esto permite que los nuevos objetos se inicialicen con constructores definidos por el usuario. Para obtener más información sobre cómo se realiza la inicialización, vea [inicializadores](#). En el ejemplo siguiente se muestra cómo usar una expresión de inicialización con el `new` operador:

```

// expre_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 mediante el `new` operador, pero no se especifica ninguna inicialización predeterminada. Por lo tanto, 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 solo se puede inicializar mediante el `new` operador si se cumple una de estas condiciones:

- Los argumentos proporcionados en el inicializador concuerdan con los 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 `new` operador; solo se llama al constructor predeterminado, si está presente. Vea [argumentos predeterminados](#) para obtener más información.

Si se produce un error en la asignación de memoria (el **operador New** devuelve un valor de 0), no se realiza ninguna inicialización. Esto 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 debe dar por hecho que estas expresiones se hayan evaluado totalmente antes de que se realice la asignación de memoria. Si se produce un error en la asignación de memoria y el `new` operador devuelve cero, es posible que algunas expresiones del inicializador no se evalúen por completo.

Duración de objetos asignados con new

Los objetos asignados con el `new` operador no se destruyen cuando se cierra el ámbito en el que se definen. Dado que el `new` operador devuelve un puntero a los objetos que asigna, el programa debe definir un puntero con el ámbito adecuado para obtener acceso a esos objetos. Por ejemplo:

```
// expe_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 expresión *de asignación* (la expresión que contiene el `new` operador) realiza tres acciones:

- 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 no es todavía 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-name* o *type-name*. El programa usa este puntero para tener acceso al objeto recién asignado.

El `new` operador invoca el operador de función **New**. Para las matrices de cualquier tipo y para los objetos que no son de `class`, `struct` o `union`, se llama a una función global, `:: Operator New`, para asignar el almacenamiento. Los objetos de tipo de clase pueden definir su propia función miembro estática **New** por clase.

Cuando el compilador encuentra el `new` operador para asignar un objeto de tipo **Type**, emite una llamada a `type :: Operator New (sizeof (type))` o, si no se define ningún operador definido por el usuario `New :: Operator New (sizeof (type))`. Por lo tanto, el `new` operador puede asignar la cantidad de memoria correcta para el objeto.

NOTE

El argumento para el **operador New** es de tipo `size_t`. Este tipo se define en `<direct.h>`, `<malloc.h>`, `<memory.h>`, `<search.h>`, `<stddef.h>`, `<stdio.h>`, `<stdlib.h>`, `<string.h>` y `<time.h>`.

Una opción en la gramática permite la especificación de la *selección de ubicación* (vea la gramática de [nuevo operador](#)). El parámetro *Placement* solo se puede usar para implementaciones definidas por el usuario de **Operator New**; permite pasar información adicional al **operador New**. Una expresión con un campo de *colocación* como `T *TObject = new (0x0040) T;` se convierte en

`T *TObject = T::operator new(sizeof(T), 0x0040);` si la clase T tiene el operador de miembro **New**, de lo contrario en `T *TObject = ::operator new(sizeof(T), 0x0040);`.

La intención original del campo de *colocación* era permitir la asignación de objetos dependientes del hardware en direcciones especificadas por el usuario.

NOTE

Aunque en el ejemplo anterior solo se muestra un argumento en el campo de *selección de ubicación*, no hay ninguna restricción en el número de argumentos adicionales que se pueden pasar al **operador nuevo** de esta manera.

Incluso cuando se ha definido **Operator New** para un tipo de clase, el operador global se puede usar con la forma de este ejemplo:

```
T *TObject = ::new TObject;
```

El operador de resolución de ámbito (`::`) fuerza el uso del `new` operador global.

Vea también

[Expresiones con operadores unarios](#)

[Palabras clave](#)

[Operadores new y delete](#)

Operador de complemento de uno: ~

02/11/2020 • 2 minutes to read • [Edit Online](#)

Sintaxis

```
~ cast-expression
```

Observaciones

El operador de complemento de uno (`~`), que a veces se denomina 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 `compl` como una ortografía alternativa para `~`. En C, la ortografía alternativa se proporciona como una macro en el `<iso646.h>` encabezado. 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++, `/permissive-` `/Za` se requiere la opción del compilador o para habilitar la ortografía alternativa.

Ejemplo

```
// 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 de enteros, vea [conversiones estándar](#).

Consulte también

[Expresiones con operadores unarios](#)

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

[Operadores aritméticos unarios](#)

Operadores de puntero a miembro: * y->*

06/03/2021 • 4 minutes to read • [Edit Online](#)

Sintaxis

```
expression .* expression
expression ->* expression
```

Observaciones

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.

```
// 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

```
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 de, y ser accesible para, el puntero a miembro especificado en el segundo operando, o bien de un tipo accesible derivado inequívocamente de y accesible para esa clase.

En una expresión que contiene el operador `-> *`, el primer operando debe ser del tipo "puntero al tipo de clase" del tipo especificado en el segundo operando, o bien debe ser de un tipo derivado de esa clase de forma no ambigua.

Ejemplo

Considere las clases y el fragmento de programa siguientes:

```
// 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.

NOTE

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.

Consulta también

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

Operadores de incremento y decremento postfijos: ++ y --

06/03/2021 • 4 minutes to read • [Edit Online](#)

Sintaxis

```
postfix-expression ++
postfix-expression --
```

Observaciones

C++ proporciona operadores de incremento y decremento tanto de prefijo como de postfijo. En esta sección, se describen solo los de postfijo. (Para obtener más información, vea [operadores de incremento y decremento prefijos](#)). La diferencia entre los dos es que, en la notación de postfijo, el operador aparece después de *postfijo-Expression*, mientras que, en la notación de prefijo, el operador aparece antes de la *expresión*. En el ejemplo siguiente, se muestra un operador de incremento de postfijo:

```
i++;
```

El efecto de aplicar el operador de incremento de postfijo (`++`) es que el valor del operando se incrementa en una unidad del tipo adecuado. Del mismo modo, el efecto de aplicar el operador de decremento de postfijo (`--`) es 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 de postfijo se evalúa como el valor de la expresión *antes* de la aplicación del operador respectivo. La operación de incremento o decremento se produce *después* de evaluarse 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 de postfijo a un puntero a una matriz de objetos de tipo, se `long` agregan cuatro a la representación interna del puntero. Este comportamiento hace que el puntero, que anteriormente hacía referencia al elemento *n* de la matriz, haga referencia al elemento (*n+ 1*) TH.

Los operandos de los operadores de incremento de postfijo y de decremento de postfijo deben ser valores l modificables (no `const`) de tipo aritmético o de puntero. El tipo del resultado es el mismo que el de la *expresión de postfijo*, pero ya no es un valor l.

Visual Studio 2017 versión 15,3 y posterior (disponible con [/STD: c++ 17](#)): el operando de un operador de incremento o decremento postfijo no puede ser de tipo `bool`.

El código siguiente muestra el operador de incremento de postfijo:

```
// 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:

```
enum Compass { North, South, East, West };
Compass myCompass;
for( myCompass = North; myCompass != West; myCompass++ ) // Error
```

Consulta también

[Expresiones de postfijo](#)

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

[Operadores de incremento y decremento postfijos de C](#)

Operadores de incremento y decremento prefijos:

++ y **--**

06/03/2021 • 3 minutes to read • [Edit Online](#)

Sintaxis

```
++ unary-expression  
-- unary-expression
```

Observaciones

El operador de incremento de prefijo (`++`) agrega uno a su operando; este valor incrementado es el resultado de la expresión. El operando debe ser un valor `I` que no sea de tipo `const`. El resultado es un valor `L` del mismo tipo que el operando.

El operador de decremento de prefijo (`--`) es análogo al operador de incremento de prefijo, salvo que el operando se reduce en uno y el resultado es este valor disminuido.

Visual Studio 2017 versión 15,3 y posterior (disponible con [/STD: c++ 17](#)): el operando de un operador de incremento o decrecimiento no puede ser 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 obtener más información, vea [operadores de incremento y decremento postfijo](#)). En el formato 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`":

```
// exre_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.

Dado que los operadores de incremento y decremento tienen efectos secundarios, el uso de expresiones con operadores de incremento o decremento en una [macro de preprocessador](#) puede tener resultados no deseados. Considere este ejemplo:

```
// expre_Increment_and_Decrement_Operators2.cpp
#define max(a,b) ((a)<(b))?(b):(a)

int main()
{
    int i = 0, j = 0, k;
    k = max( ++i, j );
}
```

La macro se expande en:

```
k = (((++i)<(j))?(j):(++i);
```

Si `i` es mayor o igual que `j` o es menor que `j` en 1, aumentará dos veces.

NOTE

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 >=

06/03/2021 • 4 minutes to read • [Edit Online](#)

Sintaxis

```
expression < expression
expression > expression
expression <= expression
expression >= expression
```

Observaciones

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; de lo contrario, el valor devuelto es `true` (1).

Ejemplo

```
// 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 secuencias (`<<`) tiene mayor prioridad que los operadores relacionales. Por consiguiente, la primera expresión sin paréntesis se evaluaría como:

```
(cout << "The true expression 3 > 2 yields: " << 3) < (2 << "\n");
```

Las conversiones aritméticas habituales descritas 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 constante que se evalúa como 0 o con un puntero de tipo `void *`. Si se realiza una comparación de puntero en 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 convierte explícitamente (Cast) al tipo `void *`. (El otro puntero se convierte implícitamente al tipo `void *` de 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 `union` y, si los dos miembros no están separados por un *especificador de acceso*, como `public`, `protected` o `private`, el puntero al miembro declarado Last se comparará más que el puntero al miembro declarado anteriormente.
- Si los dos miembros están separados por un *especificador de acceso*, los resultados son indefinidos.
- Si el tipo de clase es `union`, los punteros a miembros de datos diferentes en esa `union` comparación son iguales.

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.

Consulta también

[Expresiones con operadores binarios](#)

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

[Operadores relacionales y de igualdad de C](#)

Operador de resolución de ámbito:

06/03/2021 • 3 minutes to read • [Edit Online](#)

El operador de resolución de ámbito `::` se usa para identificar y eliminar la ambigüedad de los identificadores utilizados en ámbitos diferentes. Para obtener más información sobre el ámbito, vea [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 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.

Usar `::` 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:

```

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.

```

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 `namespace` o identificar un espacio de nombres que nombra el espacio de nombres del miembro en una `using` Directiva. En el ejemplo siguiente, puede usar `NamespaceC` para calificar `ClassB`, aunque `ClassB` se haya declarado en el espacio de nombres `NamespaceB` porque `NamespaceB` fue nombrado `NamespaceC` por una `using` Directiva.

```

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`.

```
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 ;
}
```

Usar `::` para miembros estáticos

Debe usar el operador de resolución de ámbito para llamar a miembros estáticos de clases.

```
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();
}
```

Usar `::` para enumeraciones con ámbito

El operador de resolución de ámbito también se usa con los valores de las declaraciones de [enumeración](#) de ámbito, como en el ejemplo siguiente:

```
enum class EnumA{
    First,
    Second,
    Third
};

int main() {
    EnumA enum_value = EnumA::First;
}
```

Vea también

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

[Espacios de nombres](#)

sizeof (Operador)

06/03/2021 • 4 minutes to read • [Edit Online](#)

Da como resultado el tamaño de su operando con respecto al tamaño del tipo `char`.

NOTE

Para obtener información sobre el `sizeof ...` operador, consulte [plantillas de puntos suspensivos y variádicas](#).

Sintaxis

```
sizeof unary-expression  
sizeof ( type-name )
```

Observaciones

El resultado del `sizeof` operador 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 `sizeof` puede ser uno de los siguientes:

- Nombre de tipo. Para usar `sizeof` con un nombre de tipo, el nombre debe ir entre paréntesis.
- Expresión. Cuando se utiliza con una expresión, `sizeof` se puede especificar con o sin paréntesis. La expresión no se evalúa.

Cuando el `sizeof` operador se aplica a un objeto de tipo `char`, produce 1. Cuando el `sizeof` operador se aplica a una matriz, produce 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 usa `sizeof`. Por ejemplo:

Ejemplo

```

#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

```

The size of a char is: 1
The length of Hello, world! is: 14
The size of the pointer is 4

```

Cuando el `sizeof` operador se aplica a un `class` `struct` tipo, 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 palabras. El resultado no corresponde necesariamente al tamaño que se calcula agregando los requisitos de almacenamiento de los miembros individuales. La opción del compilador `/ZP` y el pragma `Pack` afectan a los límites de alineación de los miembros.

El `sizeof` operador nunca produce 0, ni siquiera para una clase vacía.

El `sizeof` operador no se puede usar con los siguientes operandos:

- Funciones. (Sin embargo, `sizeof` se puede aplicar a punteros a funciones).
- Campos de bits.
- Clases no definidas.
- Tipo `void`.
- Matrices asignadas dinámicamente.
- Matrices externas.
- Tipos incompletos.
- Nombres entre paréntesis de tipos incompletos.

Cuando el `sizeof` operador se aplica a una referencia, el resultado es el mismo que si se `sizeof` hubiera aplicado 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 `sizeof` operador se usa a menudo para calcular el número de elementos de una matriz mediante una expresión con el formato:

```
sizeof array / sizeof array[0]
```

Vea también

[Expresiones con operadores unarios](#)

[Palabras clave](#)

Operador de subíndice []

06/03/2021 • 6 minutes to read • [Edit Online](#)

Sintaxis

```
postfix-expression [ expression ]
```

Observaciones

Una expresión de postfijo (que también puede ser una expresión primaria) seguida del operador de subíndice, `[]`, especifica la indización de la matriz.

Para obtener información sobre las matrices administradas en C++/CLI, consulte [matrices](#).

Normalmente, el valor representado por *postfijo-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. Por lo tanto, el valor entero podría estar en la posición de la *expresión de postfijo* y el valor del puntero podría estar en los corchetes de la posición de la *expresión* o subíndice. Observe el fragmento de código siguiente:

```
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]` lo proporciona:

```
*((e2) + (e1))
```

La dirección que proporciona la expresión no es de *E2* bytes de la dirección *E1*. En su lugar, la dirección se escala para producir el siguiente objeto en la matriz *E2*. Por ejemplo:

```
double aDb1[2];
```

Las direcciones de `aDb[0]` y `aDb[1]` tienen una separación de 8 bytes: el tamaño de un objeto de tipo `double`. Este ajuste de escala según el tipo de objeto lo hace automáticamente el lenguaje C++ y se define en [operadores aditivos](#) donde se describe la suma y resta de operandos de tipo de puntero.

Una expresión de subíndice también puede tener varios subíndices, como se indica a continuación:

`expression1[expresión2] [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 se evalúe la última expresión de subíndice, a menos que el valor del puntero final se convierta en 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:

```
// expe_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:

```
#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 producir un error en tiempo de ejecución porque apunta a una dirección 256 `int` posiciones inferiores en la memoria que el origen de la matriz. El puntero `midArray` se inicializa en medio de `intArray`; por lo tanto, es posible (pero peligroso) usar índices de matriz positivos y negativos en él. Los errores de subíndice de matriz no generan errores en tiempo de compilación, pero producen resultados imprevisibles.

El operador de subíndice es conmutativo. Por lo tanto, se garantiza que las expresiones `array[index]` e `index[array]` son equivalentes siempre que el operador subíndice no esté sobrecargado (vea [operadores sobrecargados](#)). La primera forma es la práctica más común de codificación, pero cualquiera de ellas funciona.

Consulta también

[Expresiones de postfijo](#)

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

[Matrices](#)

[Matrices unidimensionales](#)

[Matrices multidimensionales](#)

typeid (Operador)

06/03/2021 • 4 minutes to read • [Edit Online](#)

Sintaxis

```
typeid(type-id)
typeid(expression)
```

Observaciones

El `typeid` operador permite determinar el tipo de un objeto en tiempo de ejecución.

El resultado de `typeid` es un `const type_info&`. El valor es una referencia a un `type_info` objeto que representa el *identificador de tipo* o el tipo de la *expresión*, dependiendo de la forma de que `typeid` se use. Para obtener más información, vea [clase type_info](#).

El `typeid` operador no funciona con tipos administrados (declaradores abstractos o instancias). Para obtener información sobre cómo obtener el `Type` de un tipo especificado, vea [typeid](#).

El `typeid` operador realiza una comprobación en tiempo de ejecución cuando se aplica a un valor `I` de un tipo de clase polimórfica, donde el tipo true 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 de subíndice (`[]`). (No es seguro usar un subíndice con un puntero a un tipo polimórfico).

Si la *expresión* apunta a un tipo de clase base, pero el objeto es realmente de un tipo derivado de esa clase base, `type_info` se obtiene una referencia para la clase derivada. La *expresión* debe apuntar a un tipo polimórfico (una clase con funciones virtuales). De lo contrario, el resultado es el `type_info` de la clase estática a la que se hace referencia en la *expresión*. Además, se debe desreferenciar el puntero para que el objeto utilizado sea el que señala. Sin desreferenciar el puntero, el resultado será el del `type_info` puntero, no lo que señala. Por ejemplo:

```

// 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 *expresión* está desreferenciando un puntero y el valor de ese puntero es cero, se `typeid` produce una excepción `bad_typeid`. Si el puntero no apunta a un objeto válido, `__non_rtti_object` se produce una excepción. Indica un intento de analizar la RTTI que desencadenó un error porque el objeto no es válido de algún modo. (Por ejemplo, es un puntero incorrecto o el código no se compiló con `/gr`).

Si la *expresión* no es un puntero, y no una referencia a una clase base del objeto, el resultado es una `type_info` referencia que representa el tipo estático de la *expresión*. El *tipo estático* de una expresión hace referencia al tipo de una expresión, ya que se conoce en 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:

```

// expre_typeid_Operator_2.cpp
#include <typeinfo>

int main()
{
    typeid(int) == typeid(int&); // evaluates to true
}

```

`typeid` también se puede usar en plantillas para determinar el tipo de un parámetro de plantilla:

```

// 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 );
}

```

Consulta también

[Información de tipo en tiempo de ejecución](#)

[Palabras clave](#)

Operadores unarios más y de negación: + y -

06/03/2021 • 2 minutes to read • [Edit Online](#)

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. Por lo tanto, la expresión `+ch` , donde `ch` es de tipo `char` , da como resultado el tipo `int` ; el valor no se modifica. Vea [conversiones estándar](#) para obtener más información sobre cómo se realiza la promoción.

- (operador)

El operador unario de negación (-) 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. Vea [conversiones estándar](#) para obtener más información sobre cómo se realiza 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++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 principales](#)
- [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 tipo en tiempo de ejecución](#)

Temas sobre operadores en otras secciones:

- [Operadores integrados de C++, precedencia y asociatividad](#)
- [Operadores sobrecargados](#)
- [typeid \(C++/CLI\)](#)

NOTE

Los operadores para los tipos integrados no se pueden sobrecargar; su comportamiento está predefinido.

Vea también

[Referencia del lenguaje C++](#)

Tipos de expresiones

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 constantes](#). 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 tipo en tiempo de ejecución](#). Determine el tipo de un objeto durante la ejecución del programa.

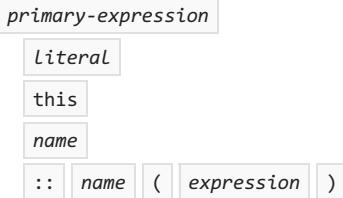
Consulta también

[Expresiones](#)

Expresiones primarias

02/11/2020 • 3 minutes to read • [Edit Online](#)

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:



Una `literal` es una expresión primaria constante. Su tipo depende de la forma de su especificación. Para obtener información completa sobre cómo especificar literales, vea [literales](#).

La `this` palabra clave es un puntero a un objeto de clase. Está disponible en funciones miembro no estáticas. Apunta a la instancia de la clase para la que se invocó la función. La `this` palabra clave no se puede usar fuera del cuerpo de una función miembro de clase.

El tipo del `this` puntero es `type * const` (donde `type` es el nombre de clase) dentro de las funciones que no modifican específicamente el `this` puntero. En el ejemplo siguiente se muestran las declaraciones de funciones miembro y los tipos de `this`:

```
// expre_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 `this` puntero, vea [this Pointer](#).

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 viene 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 declarativo 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. Vea [ámbito](#) para obtener un ejemplo de cómo usar el operador de resolución de ámbito.

Una expresión entre paréntesis es una expresión primaria. Su tipo y valor son idénticos al tipo y valor de la expresión no entre paréntesis. Es un valor l si la expresión no entre paréntesis es un valor l.

Entre los ejemplos de expresiones primarias se incluyen:

```
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
```

En estos ejemplos se consideran *nombres*, como tales, expresiones principales, en varios formatos:

```
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

06/03/2021 • 6 minutes to read • [Edit Online](#)

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 resulta especialmente útil para los desarrolladores de bibliotecas de C++, ya que puede aplicarse tanto a las plantillas de clase como a las plantillas de función y, por tanto, proporciona una gama amplia de funcionalidad y flexibilidad con seguridad de tipos y no triviales.

Sintaxis

Las plantillas variádicas utilizan los puntos suspensivos de dos maneras. 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.

A continuación se muestra un ejemplo básico de sintaxis de definición de *clase de plantilla variádica*:

```
template<typename... Arguments> class classname;
```

Tanto en los paquetes de parámetros como en las expansiones se puede agregar espacio en blanco alrededor de los puntos suspensivos, según se deseé, como se muestra en estos ejemplos:

```
template<typename ...Arguments> class classname;
```

O bien:

```
template<typename ... Arguments> class classname;
```

Tenga en cuenta que 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:

```
template<typename... Arguments> class vtclass;  
  
vtclass< > vtinstance1;  
vtclass<int> vtinstance2;  
vtclass<float, bool> vtinstance3;  
vtclass<long, std::vector<int>, std::string> vtinstance4;
```

Cuando se usa una definición de clase de plantilla variádica, también puede ser necesario al menos un parámetro:

```
template <typename First, typename... Rest> class classname;
```

A continuación se muestra un ejemplo básico de sintaxis de *función de plantilla variádicas*:

```
template <typename... Arguments> returntype functionname(Arguments... args);
```

A continuación, el paquete de parámetros *arguments* se expande para su uso, como se muestra en la sección siguiente, **Descripción de las plantillas variádicas**.

Es posible utilizar otras formas de sintaxis de función de plantilla variática, incluidas las de estos ejemplos, entre otras:

```
template <typename... Arguments> returntype functionname(Arguments&... args);
template <typename... Arguments> returntype functionname(Arguments&&... args);
template <typename... Arguments> returntype functionname(Arguments*... args);
```

`const` También se permiten especificadores como:

```
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:

```
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()`):

```
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 ha descrito la posición de los puntos suspensivos que definen los paquetes de parámetros y las expansiones como "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". Esto es técnicamente cierto pero puede ser confuso trasladarlo 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" presentan un paquete de parámetros de función y la posición de los puntos suspensivos es importante:

```
// 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 mostrar el mecanismo de la función de plantilla variádica consiste en utilizarlo para reescribir cierta funcionalidad de `printf`:

```
#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);  
}
```

Output

```
1  
10, 20  
100, 200, 300  
first, 2, third, 3.14159
```

NOTE

La mayoría de las implementaciones que incorporan funciones de plantilla variádicas utilizan la recursividad de alguna forma, pero es ligeramente diferente de la recursividad tradicional. La recursividad tradicional conlleva una función que se llama a sí misma utilizando la misma firma. (Puede estar sobrecargado o con plantilla, pero se elige la misma firma cada vez). La recursividad variádicas implica llamar a una plantilla de función variádicas mediante el uso de números de argumentos diferentes (casi siempre en disminución) y, por lo tanto, marcan una firma diferente cada vez. Sigue siendo necesario un "caso base", pero la naturaleza de la recursividad es diferente.

Expresiones postfijas

06/03/2021 • 12 minutes to read • [Edit Online](#)

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>type-name</i> ()
Operador de acceso a miembros	. de ->
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* anterior 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:

```
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 toda la expresión 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. A continuación, llame a `GetValue()` en esa clase y, a continuación, incremente 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

```
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 es uno de `dynamic_cast`, `static_cast` o `reinterpret_cast`. Puede encontrar más información en [dynamic_cast](#), [static_cast](#) y [reinterpret_cast](#).

El `typeid` operador se considera una expresión de postfijo. Vea [operador typeid](#).

Argumentos formales y reales

Los programas que llaman pasan información a las funciones llamadas en "argumentos reales". Las funciones llamadas tienen acceso a la información mediante el uso de los correspondientes "argumentos formales".

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 inicializara; tanto las conversiones estándar como las definidas por el usuario se realizan en la conversión de un argumento real al tipo correcto. Inicialización realizada se muestra de forma conceptual en el código siguiente:

```
void Func( int i ); // Function prototype
...
Func( 7 );           // Execute function call
```

Las inicializaciones conceptuales anteriores a la llamada son:

```
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, vea [inicializadores](#) y [conversiones](#)).

Por consiguiente, si el prototipo de función (declaración) llama a para un argumento de tipo `long`, y si el programa que realiza la llamada proporciona un argumento real de tipo `int`, el argumento real se promueve utilizando una conversión de tipo estándar al tipo `long` (vea [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. (Vea [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:

```

// expre_Formal_and_Actual_Arguments.cpp
void func( long param1, double param2 );

int main()
{
    long i = 1;
    double j = 2;

    // Call func with actual arguments i and j.
    func( i, j );
}

// Define func with formal parameters param1 and param2.
void func( long param1, double param2 )
{
}

```

Cuando `func` se llama a desde Main, el parámetro formal `param1` se inicializa con el valor de `i` (`i` se convierte al tipo `long` para que se corresponda con el 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 `const` tipos 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 fuera una referencia a un objeto que no sea de tipo `const`.

Las funciones siguientes muestran algunos de estos conceptos:

```

// 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 indican que los argumentos pueden ser necesarios pero que el número y los tipos no se especifican 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. Las distintas conversiones se aplican a las funciones declaradas con puntos suspensivos que a las funciones para las que se conocen los tipos de argumentos formales y reales:

- Si el argumento real es de tipo `float`, se promueve al tipo `double` antes de la llamada de función.

- Cualquier `signed char` o, o, un `unsigned char` `signed short` `unsigned short` tipo enumerado o un campo de bits se convierte en `signed int` O `unsigned int` mediante promoción de entero.
- 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 utilizan, 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 con CLR, vea [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, vea [argumentos predeterminados](#).

```
// 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

```
hello,
world!
good morning, sunshine.
```

Consulta también

Tipos de expresiones

Expresiones con operadores unarios

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 de prefijo (++)
- Operador de decremento de prefijo (--)
- Operador de conversión ()
- `sizeof` Operator
- `__uuidof` Operator
- `alignof` Operator
- `new` Operator
- `delete` Operator

Estos operadores tienen asociatividad de derecha a izquierda. Las expresiones unarias normalmente usan sintaxis que precede a una expresión de postfijo o primaria.

Estas son las posibles formas de expresiones unarias.

- *postfix-expression*
- `++ unary-expression`
- `-- unary-expression`
- *unario- conversión de operador-expresión*
- `* sizeof ***unario-expresión`
- `sizeof(nombre de tipo)`
- `decltype(expresión de)`
- *asignación: expresión*
- *desasignación: expresión*

Cualquier *expresión de postfijo* se considera una *expresión unaria*, y dado que cualquier expresión principal se considera una *expresión de postfijo*, cualquier expresión principal se considera también una *expresión unaria*.

Para obtener más información, vea [expresiones de postfijo](#) y [expresiones primarias](#).

Un *operador unario* se compone de uno o varios de los siguientes símbolos: `* & + - ! ~`

Cast-Expression es una expresión unaria con una conversión opcional para cambiar el tipo. Para obtener más información, vea [operador de conversión: \(\)](#).

Una *expresión* puede ser cualquier expresión. Para obtener más información, vea [expresiones](#).

La *expresión de asignación* hace referencia al `new` operador. La *cancelación de asignación de la expresión* hace referencia al `delete` operador. Para obtener más información, vea los vínculos anteriores en este tema.

Consulta también

[Tipos de expresiones](#)

Expresiones con operadores binarios

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 de resta (-=)

- Asignación y multiplicación (*=)
- Asignación y división (/=)
- Asignación y módulo (%=)
- Asignación de desplazamiento a la izquierda (<<=)
- Asignación de desplazamiento a la derecha (>>=)
- Asignación and bit a bit (&=)
- Asignación y OR exclusivo bit a bit (^=)
- Asignación or inclusivo bit a bit (|=)
- [Operador de coma \(,\)](#)

Consulta también

[Tipos de expresiones](#)

Expresiones constantes de C++

06/03/2021 • 2 minutes to read • [Edit Online](#)

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
- `sizeof` Expresiones

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.

```
const double Size = 11.0;
char chArray[(int)Size];
```

Las conversiones explícitas a tipos enteros son legales en expresiones constantes; el resto de tipos y los tipos derivados no son válidos excepto cuando se utilizan como operandos para el `sizeof` operador.

El operador de coma y los operadores de asignación no se pueden utilizar en expresiones constantes.

Consulta también

[Tipos de expresiones](#)

Semántica de las expresiones

06/03/2021 • 9 minutes to read • [Edit Online](#)

Las expresiones se evalúan según la prioridad y agrupación de sus operadores. ([Precedencia y asociatividad](#) de los operadores en las [convenciones léxicas](#), muestra las relaciones que imponen los operadores de C++ en las expresiones).

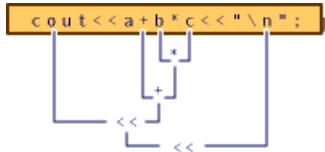
Orden de evaluación

Considere este ejemplo:

```
// 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";
}
```

38
38
54

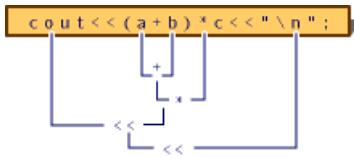


Orden de evaluación de expresiones

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 hay dos repeticiones. 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 evaluación de expresiones 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
<i>type</i>	* <code>const</code> *** <i>tipo</i> de * <code>volatile</code> *** <i>tipo</i> de <i>type&</i> * <code>const</code> *** <i>tipo</i> de& * <code>volatile</code> *** <i>tipo</i> de& <code>volatile const</code> <i>tipo</i> de <code>volatile const</code> <i>tipo</i> de&
<i>tipo</i> de *	<i>tipo</i> de * * <code>const</code> *** <i>tipo</i> de* * <code>volatile</code> *** <i>tipo</i> de* <code>volatile const</code> <i>tipo</i> de*
* <code>const</code> *** <i>tipo</i> de	<i>type</i> * <code>const</code> *** <i>tipo</i> de * <code>const</code> *** <i>tipo</i> de&
* <code>volatile</code> *** <i>tipo</i> de	<i>type</i> * <code>volatile</code> *** <i>tipo</i> de * <code>volatile</code> *** <i>tipo</i> de&

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 lógico OR (`||`). 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.

Consulta también

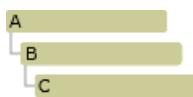
[Expresiones](#)

Conversión

06/03/2021 • 3 minutes to read • [Edit Online](#)

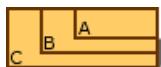
El lenguaje C++ permite que, si una clase se deriva de una clase base que contiene funciones virtuales, se pueda usar un puntero a ese tipo de clase base para llamar a las implementaciones de las funciones virtuales que residen en el objeto de la clase derivada. Una clase que contiene funciones virtuales 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" y que el puntero a la clase base apunta a un "subobjeto" del objeto completo. Consideré, 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 no polimorfismo, 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 tipo en tiempo de ejecución](#)

Consulta también

[Expresiones](#)

Operadores de conversión

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 polimorfismo.
- [const_cast](#) Se usa para quitar `const` los `volatile` atributos, y `__unaligned`.
- [reinterpret_cast](#) Se utiliza para la reinterpretación simple de bits.
- [safe_cast](#) Se usa en C++/CLI para generar MSIL comprobable.

Use `const_cast` y `reinterpret_cast` como último recurso, ya que estos operadores presentan los mismos peligros que las conversiones de estilo antiguas. Sin embargo, siguen siendo necesarios para reemplazar completamente las conversiones antiguas.

Consulta también

[Conversión](#)

dynamic_cast (Operador)

02/11/2020 • 12 minutes to read • [Edit Online](#)

Convierte el operando `expression` en un objeto del tipo `type-id`.

Sintaxis

```
dynamic_cast < type-id > ( expression )
```

Observaciones

`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.

Vea [static_cast](#) para obtener una explicación de la diferencia entre las conversiones estáticas y dinámicas, y Cuándo es adecuado utilizar cada una de ellas.

Hay dos cambios importantes en el comportamiento de `dynamic_cast` en código administrado:

- `dynamic_cast` para un puntero al tipo subyacente de una enumeración con conversión Boxing se producirá un error en tiempo de ejecución, devolviendo 0 en lugar del puntero convertido.
- `dynamic_cast` ya no producirá una excepción cuando `type-id` sea un puntero interior a un tipo de valor, con la conversión errónea 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:

```
// 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:

```

// 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:

```

// 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` da como resultado una operación no operativa si la conversión se puede realizar implícitamente, o una `isinst` instrucción MSIL, que realiza una comprobación dinámica y devuelve `nullptr` si se produce un error en la conversión.

En el ejemplo siguiente `dynamic_cast` se utiliza para determinar si una clase es una instancia de un tipo determinado:

```

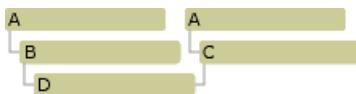
// dynamic_cast_clr.cpp
// compile with: /clr
using namespace System;

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:

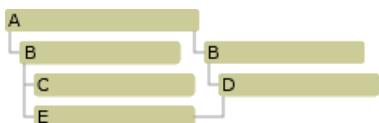
```

// 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
}

```

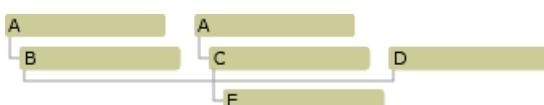
Se pueden introducir más ambigüedades cuando se utilizan clases base virtuales. Considere la jerarquía de clases que se muestra en la ilustración siguiente.



Jerarquía de clases que muestra clases base virtuales

En esta jerarquía, **A** es una clase base virtual. Dada una instancia de **E** la clase y un puntero al **A** subobjeto, **dynamic_cast** se producirá un error en un a un puntero a, **B** 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. Puede realizar una `dynamic_cast` conversión del `D` puntero a un `E` puntero, una conversión (`dynamic_cast` o una conversión implícita) de `E` a `B` y, por último, una conversión implícita de `B` a `A`. Por ejemplo:

```
// 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 `dynamic_cast` operador también se puede usar para realizar una "conversión cruzada". Utilizando la misma jerarquía de clases, es posible convertir un puntero (por ejemplo, del subobjeto `B` al subobjeto `D`) siempre y cuando 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:

```
// 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 NULL 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:

```
// 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. Un error de conversión al tipo de referencia produce una [excepción bad_cast](#). Si no `expression` señala ni hace referencia a un objeto válido, `__non_rtti_object` se produce una excepción.

Vea [typeid](#) para obtener una explicación de la `__non_rtti_object` excepción.

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.

```

// 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);
}

```

```
in C
test2 in B
in GlobalTest
Can't cast to C
```

Consulte también

[Operadores de conversión](#)

[Palabras clave](#)

bad_cast (Excepción)

06/03/2021 • 2 minutes to read • [Edit Online](#)

El operador produce la excepción **bad_cast** `dynamic_cast` como resultado de una conversión errónea a un tipo de referencia.

Sintaxis

```
catch (bad_cast)
    statement
```

Observaciones

La interfaz para **bad_cast** es:

```
class bad_cast : public exception
```

El código siguiente contiene un ejemplo de un error `dynamic_cast` que produce la excepción **bad_cast**.

```
// 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();
    }
}
```

La excepción se produce porque el objeto que se va a convertir (una forma) no se deriva del tipo de conversión especificado (círculo). Para evitar la excepción, agregue estas declaraciones a `main`:

```
Circle circle_instance;
Circle& ref_circle = circle_instance;
```

A continuación, invierta el sentido de la conversión en el bloque de la `try` siguiente manera:

```
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>elemento</code>	TBD

Operadores

OPERATOR	DESCRIPCIÓN
<code>operador =</code>	Operador de asignación que asigna un <code>bad_cast</code> objeto a otro.

bad_cast

Constructor para los objetos de tipo `bad_cast`.

```
bad_cast(const char * _Message = "bad cast");
bad_cast(const bad_cast &);
```

operador =

Operador de asignación que asigna un `bad_cast` objeto a otro.

```
bad_cast& operator=(const bad_cast&) noexcept;
```

elemento

```
const char* what() const noexcept override;
```

Consulta también

[dynamic_cast \(operador\)](#)

[Palabra](#)

[Procedimientos recomendados de C++ moderno para excepciones y control de errores](#)

static_cast (Operador)

06/03/2021 • 7 minutes to read • [Edit Online](#)

Convierte una *expresión* al tipo de identificador de *tipo*, basándose solo en los tipos que se encuentran en la expresión.

Sintaxis

```
static_cast <type-id> ( expression )
```

Observaciones

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](#).

El `static_cast` operador se puede utilizar para operaciones como la conversión de un puntero a una clase base en un puntero a una clase derivada. Estas conversiones no siempre son seguras.

En general, se usa `static_cast` cuando se desea convertir tipos de datos numéricos, como las enumeraciones, a ints o ints en valores Float, y está seguro de los tipos de datos implicados en la conversión. `static_cast` las conversiones no son tan seguras como las `dynamic_cast` conversiones, porque no `static_cast` realiza ninguna comprobación de tipo en tiempo de ejecución, mientras que `dynamic_cast` sí. Un `dynamic_cast` a un puntero ambiguo producirá un error, mientras `static_cast` que devuelve como si no hubiera ningún problema; esto puede ser peligroso. Aunque `dynamic_cast` las conversiones son más seguras, `dynamic_cast` solo funcionan en punteros o referencias, y la comprobación de tipo en tiempo de ejecución es una sobrecarga. Para obtener más información, vea [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 `static_cast` conversión 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 `dynamic_cast` `static_cast` operadores y mueven un puntero a lo largo de una jerarquía de clases. Sin embargo, `static_cast` confía exclusivamente en la información proporcionada en la instrucción de conversión y, por tanto, no puede ser segura. Por ejemplo:

```
// 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 a 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 la `D` clase completa, `dynamic_cast` sabrá lo suficiente como para devolver cero. Sin embargo, `static_cast` se basa en la asercción del programador que `pb` apunta a un objeto de tipo `D` y simplemente devuelve un puntero a ese `D` objeto supuesto.

Por consiguiente, `static_cast` puede hacer el inverso de las conversiones implícitas, en cuyo caso los resultados son indefinidos. Se deja al programador comprobar que los resultados de una `static_cast` conversión son seguros.

Este comportamiento también se aplica a los tipos distintos de los tipos de clase. Por ejemplo, `static_cast` se puede utilizar para convertir de `int` a `char`. Sin embargo, es posible que el resultado `char` no tenga suficientes bits para contener el `int` valor completo. Una vez más, se deja al programador comprobar que los resultados de una `static_cast` conversión son seguros.

El `static_cast` operador también se puede utilizar para realizar cualquier conversión implícita, incluidas las conversiones estándar y las conversiones definidas por el usuario. Por ejemplo:

```
// 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 `static_cast` operador 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 `static_cast` operador convierte un valor de puntero nulo en el valor de puntero nulo del tipo de destino.

Cualquier expresión se puede convertir explícitamente al tipo `void` por el `static_cast` operador. El tipo `void` de destino puede incluir opcionalmente `const` el `volatile` atributo, o `__unaligned`.

El `static_cast` operador no puede desechar `const`, los `volatile` atributos, o `__unaligned`. Consulte [Const_cast operador](#) para obtener información sobre cómo quitar estos atributos.

C++/CLI: Debido al riesgo de realizar conversiones no comprobadas sobre un recolector de elementos no utilizados de reubicación, el uso de `static_cast` solo debe estar en código crítico para el rendimiento cuando esté seguro de que funcionará correctamente. Si debe usar `static_cast` en modo de versión, sustituya por `safe_cast` en las compilaciones de depuración para asegurarse de que se ha realizado correctamente.

Consulta también

[Operadores de conversión](#)

[Palabras clave](#)

const_cast (Operador)

06/03/2021 • 2 minutes to read • [Edit Online](#)

Quita los `const` `volatile` atributos, y `__unaligned` de una clase.

Sintaxis

```
const_cast <type-id> (expression)
```

Observaciones

Un puntero a cualquier tipo de objeto o un puntero a un miembro de datos se puede convertir explícitamente en un tipo que es idéntico `const`, salvo los `volatile` `__unaligned` calificadores, y. 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 se puede usar el `const_cast` operador para invalidar directamente el estado constante de una variable de constante.

El `const_cast` operador convierte un valor de puntero nulo en el valor de puntero nulo del tipo de destino.

Ejemplo

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

using namespace std;
class CCTest {
public:
    void setNumber( int );
    void printNumber() const;
private:
    int number;
};

void CCTest::setNumber( int num ) { number = num; }

void CCTest::printNumber() const {
    cout << "\nBefore: " << number;
    const_cast< CCTest * >( this )->number--;
    cout << "\nAfter: " << number;
}

int main() {
    CCTest X;
    X.setNumber( 8 );
    X.printNumber();
}
```

En la línea que contiene `const_cast`, el tipo de datos del `this` puntero es `const CCTest *`. El `const_cast`

operador cambia el tipo de datos del `this` puntero a `cctest *`, lo que permite que `number` se modifique el miembro. La conversión se produce únicamente para el resto de la instrucción en la que aparece.

Consulta también

[Operadores de conversión](#)

[Palabras clave](#)

reinterpret_cast (Operador)

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 )
```

Observaciones

El uso incorrecto del `reinterpret_cast` operador puede ser fácilmente no seguro. A menos que la conversión deseada sea inherentemente de bajo nivel, se debe utilizar uno de los otros operadores de conversión.

El `reinterpret_cast` operador se puede utilizar para las conversiones como `char*` `int*`, o `One_class*` en `Unrelated_class*`, que son intrínsecamente no seguras.

El resultado de `reinterpret_cast` no se puede usar de forma segura para ningún elemento que no sea una conversión a su tipo original. Otros usos son, en el mejor de los casos, no portables.

El `reinterpret_cast` operador no puede desechar `const` los `volatile` atributos, o `__unaligned`. Consulte [Const_cast operador](#) para obtener información sobre cómo quitar estos atributos.

El `reinterpret_cast` operador convierte un valor de puntero nulo en el valor de puntero nulo del tipo de destino.

Un uso práctico de `reinterpret_cast` está en una función hash, que asigna un valor a un índice de tal forma que dos valores distintos raramente terminan con el mismo índice.

```
#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 trate como un 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.

Consulta también

[Operadores de conversión](#)

[Palabras clave](#)

Información de tipos en tiempo de ejecución

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 utiliza para contener la información de tipo devuelta por el `typeid` operador.

Consulta también

[Conversión](#)

bad_typeid (Excepción)

06/03/2021 • 2 minutes to read • [Edit Online](#)

El operador typeid produce la excepción **bad_typeid** cuando el operando de typeid es un puntero nulo.

Sintaxis

```
catch (bad_typeid)
    statement
```

Observaciones

La interfaz para bad_typeid es:

```
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 typeid operador que produce una excepción **bad_typeid**.

```
// 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

Object is NULL

Vea también

[Información de tipo en tiempo de ejecución](#)

[Palabras clave](#)

type_info (Clase)

06/03/2021 • 4 minutes to read • [Edit Online](#)

En la clase `type_info` se 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 de igualdad o de ordenación. Las reglas de codificación y la secuencia de intercalación para tipos no se especifican y pueden diferir entre programas.

El `<typeinfo>` archivo de encabezado debe estar incluido para poder usar la clase `type_info`. La interfaz para la clase `type_info` es:

```
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 se pueden crear instancias de objetos de la clase `type_info` directamente, porque la clase solo tiene un constructor de copias privado. La única manera de construir un objeto de `type_info` (temporal) es usar el operador `typeid`. Dado que el operador de asignación también es privado, no puede copiar o 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 `type_info::before` función miembro para determinar la secuencia de intercalación de tipos. No hay ninguna garantía que `type_info::before` produzca el mismo resultado en programas diferentes o incluso en diferentes ejecuciones del mismo programa. De esta manera, `type_info::before` es similar al operador Address-of `(&)`.

La `type_info::name` función miembro devuelve `const char*` a una cadena terminada 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 `type_info::raw_name` función miembro devuelve `const char*` a una cadena terminada 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 es necesario quitar el decorado del nombre. La cadena devuelta por la `type_info::raw_name` función es útil en las operaciones de comparación pero no es legible. Si necesita una cadena inteligible para el usuario, utilice `type_info::name` en su lugar la función.

La información de tipo se genera para las clases polimórficas solo si se especifica la opción de compilar `/gr`

(habilitar información de tipo Run-Time) .

Consulta también

[Información de tipos en tiempo de ejecución](#)

Instrucciones (C++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

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++

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 `declaration-statement`, que realmente quita 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

06/03/2021 • 5 minutes to read • [Edit Online](#)

Las etiquetas se usan para transferir el control de programas directamente a la instrucción especificada.

```
identifier : statement
case constant-expression : statement
default : statement
```

El ámbito de una etiqueta es toda la función donde se declara.

Observaciones

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. La etiqueta case y las etiquetas predeterminadas son específicas para las instrucciones case.

```
#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
}
```

La instrucción goto

La apariencia de una etiqueta de *identificador* en el programa de origen declara una etiqueta. Solo una instrucción `goto` puede transferir el control a una etiqueta de *identificador*. En el fragmento de código siguiente se muestra el uso de la `goto` instrucción y una etiqueta de *identificador*:

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.

```

// 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.

```

La instrucción case

Las etiquetas que aparecen después de la `case` palabra clave también no pueden aparecer fuera de una `switch` instrucción. (Esta restricción también se aplica a la `default` palabra clave). En el fragmento de código siguiente se muestra el uso correcto de las `case` Etiquetas:

```

// 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;

    default:
        // This choice is taken for all messages not specifically
        // covered by a case statement.

        return DefWindowProc( hWnd, Message, wParam, lParam );
    break;
}

```

Etiquetas en la instrucción case

Las etiquetas que aparecen después de la `case` palabra clave también no pueden aparecer fuera de una `switch` instrucción. (Esta restricción también se aplica a la `default` palabra clave). En el fragmento de código siguiente se muestra el uso correcto de las `case` Etiquetas:

```

// 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:
        // Obtain a handle to the device context.
        // BeginPaint will send WM_ERASEBKGND if appropriate.

        memset( &ps, 0x00, sizeof(PAINTSTRUCT) );
        hDC = BeginPaint( hWnd, &ps );

        // Inform Windows that painting is complete.

        EndPaint( hWnd, &ps );
        break;

    case WM_CLOSE:
        // Close this window and all child windows.

        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;
}

```

Etiquetas en la instrucción goto

La apariencia de una etiqueta de *identificador* en el programa de origen declara una etiqueta. Solo una instrucción **goto** puede transferir el control a una etiqueta de *identificador*. En el fragmento de código siguiente se muestra el uso de la **goto** instrucción y una etiqueta de *identificador*:

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.

```
// 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;
    // At Test2 label.
}
```

Consulta también

[Información general sobre las instrucciones de C++](#)

[Switch \(instrucción\) \(C++\)](#)

Expression (Instrucción)

06/03/2021 • 2 minutes to read • [Edit Online](#)

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] ;
```

Observaciones

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. Dado que la expresión es opcional, un punto y coma solo se considera una instrucción de expresión vacía, que se conoce como la instrucción `null`.

Consulta también

[Información general sobre las instrucciones de C++](#)

NULL (Instrucción)

06/03/2021 • 2 minutes to read • [Edit Online](#)

La "instrucción null" es una instrucción de expresión con la *expresión* que falta. 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:

```
// 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()
{}
```

Consulta también

[Expression \(instrucción\)](#)

Instrucciones compuestas (Bloques)

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 ] }
```

Observaciones

En el ejemplo siguiente se usa una instrucción compuesta *como parte de la instrucción de* la instrucción `if` (vea [la instrucción If](#) para obtener más información sobre la sintaxis):

```
if( Amount > 100 )
{
    cout << "Amount was too large to handle\n";
    Alert();
}
else
{
    Balance -= Amount;
}
```

NOTE

Dado que una declaración es una instrucción, una declaración puede ser una de las instrucciones de la *lista de instrucciones*. Por consiguiente, los nombres declarados dentro de una instrucción compuesta, pero no declarados explícitamente como static, tienen ámbito local y (para objetos) duración. Vea [ámbito](#) para obtener más información sobre el tratamiento de nombres con ámbito local.

Consulta también

[Información general sobre las instrucciones de C++](#)

Instrucciones de selección (C++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

Las instrucciones de selección de C++, [si](#) y [Switch](#), proporcionan un medio para ejecutar condicionalmente secciones de código.

Las instrucciones [__if_exists](#) y [__if_not_exists](#) permiten incluir código condicionalmente dependiendo de la existencia de un símbolo.

Vea en cada tema individual la sintaxis de cada instrucción.

Consulta también

[Información general sobre las instrucciones de C++](#)

IF-Else (instrucción) (C++)

02/11/2020 • 5 minutes to read • [Edit Online](#)

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 opcional `else`. De lo contrario, se omite la siguiente instrucción y, si hay una `else`, se ejecuta la instrucción que sigue a `else`.

`condition` las expresiones que se evalúan como distintas de cero son:

- `true`
- un puntero no nulo,
- cualquier valor aritmético distinto de cero, o bien
- tipo de clase que define una conversión no ambigua a un tipo aritmético, booleano o de puntero. (Para obtener información sobre las conversiones, vea [conversiones estándar](#)).

Sintaxis

```
init-statement :  
    expression-statement  
    simple-declaration  
  
condition :  
    expression  
    attribute-specifier-seq opt decl-specifier-seq OPT declarator ** brace-or-equal-initializer  
  
statement :  
    expression-statement  
    compound-statement  
  
expression-statement :  
    expression OPT ;  
  
compound-statement :  
    { statement-seq OPT }  
  
statement-seq :  
    statement  
    statement-seq statement  
  
if-branch :  
    statement  
  
else-branch :  
    statement  
  
selection-statement :  
    if constexpr OPT17 ( init-statement opt17 OPC condition * ) *** if-branch  
    if constexpr OPT17 ( init-statement opt17 OPC condition ) if-branch * else *** else-branch
```

¹⁷ este elemento opcional está disponible a partir de c++ 17.

instrucciones IF-Else

En todas las formas de la `if` instrucción, `condition`, que puede tener cualquier valor excepto una estructura, se evalúa, incluidos todos los efectos secundarios. El control pasa de la `if` instrucción a la siguiente instrucción del programa, a menos que se ejecute `if-branch` o `else-branch` contenga un `break`, `continue` o `goto`.

La `else` cláusula de una `if...else` instrucción está asociada a la instrucción anterior más cercana `if` en el mismo ámbito que no tiene una `else` instrucción correspondiente.

Ejemplo

En este código de ejemplo se muestran varias `if` instrucciones en uso, con y sin `else`:

```
// 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 `if` instrucción también puede contener una `init-statement` expresión que declara e inicializa una variable con nombre. Utilice este formulario de la instrucción If-cuando la variable solo sea necesaria en el ámbito de la instrucción If-Statement. **Específico de Microsoft:** este formulario está disponible a partir de la versión 15,3 de Visual Studio 2017 y requiere al menos la `/std:c++17` opción del compilador.

Ejemplo

```
#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";
    }
}
```

if (instrucciones) de constexpr

A partir de C++ 17, puede usar una `if constexpr` instrucción en las 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 `/std:c++17` opción del compilador.

Ejemplo

En este ejemplo se muestra cómo se puede escribir una función única que controle el desempaquetado de parámetros. No se necesita ninguna sobrecarga de parámetros cero:

```
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](#)

[Palabra](#)

[Instrucción `switch` \(C++\)](#)

`__if_exists` (Instrucción)

06/03/2021 • 2 minutes to read • [Edit Online](#)

La `__if_exists` instrucción comprueba 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.

afirma

Una o varias instrucciones que se ejecutarán si el *identificador* existe.

Observaciones

Caution

Para obtener los resultados más confiables, utilice la `__if_exists` instrucción bajo las siguientes restricciones.

- Aplique la `__if_exists` instrucción solo a tipos simples, no a plantillas.
- Aplique la `__if_exists` instrucción a los identificadores tanto dentro como fuera de una clase. No aplique la `__if_exists` instrucción a las variables locales.
- Use la `__if_exists` instrucción solo en el cuerpo de una función. Fuera del cuerpo de una función, la `__if_exists` instrucción 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 `__if_exists` instrucción es la instrucción `__if_not_exists`.

Ejemplo

Observe que este ejemplo utiliza plantillas, lo que no se recomienda.

```

// 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

```

In X<T>::Dump()
In A::Dump()
In X<T>::Dump()
T::Dump does not exist
g_bFlag = 1
C::f exists

```

Vea también

[Instrucciones de selección](#)

[Palabras clave](#)

[Instrucción __if_not_exists](#)

`_if_not_exists` (Instrucción)

06/03/2021 • 2 minutes to read • [Edit Online](#)

La `_if_not_exists` instrucción comprueba 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.

afirma

Una o varias instrucciones que se ejecutarán si el *identificador* no existe.

Observaciones

Caution

Para obtener los resultados más confiables, utilice la `_if_not_exists` instrucción bajo las siguientes restricciones.

- Aplique la `_if_not_exists` instrucción solo a tipos simples, no a plantillas.
- Aplique la `_if_not_exists` instrucción a los identificadores tanto dentro como fuera de una clase. No aplique la `_if_not_exists` instrucción a las variables locales.
- Use la `_if_not_exists` instrucción solo en el cuerpo de una función. Fuera del cuerpo de una función, la `_if_not_exists` instrucción 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 `_if_not_exists` instrucción es la instrucción `_if_exists`.

Ejemplo

Para obtener un ejemplo sobre cómo usar `_if_not_exists`, vea la [instrucción `_if_exists`](#).

Consulta también

[Instrucciones de selección](#)

[Palabras clave](#)

[Instrucción `_if_exists`](#)

switch Statement (C++)

06/03/2021 • 8 minutes to read • [Edit Online](#)

Permite la selección entre varias secciones de código, dependiendo del valor de una expresión entera.

Sintaxis

```
selection-statement :  
    switch ( [ init-statement ] opt C++ 17 condition ) statement
```

```
init-statement :  
    expression-statement  
    simple-declaration
```

```
condition :  
    expression  
    attribute-specifier-seq opt 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 un tipo de clase que tenga una conversión no ambigua a un tipo entero. La promoción de entero tiene lugar tal y como se describe en [conversiones estándar](#).

El `switch` cuerpo de la instrucción consta de una serie de `case` etiquetas y una opt `default` etiqueta ional. Una `Labeled-statement` es una de estas etiquetas y las instrucciones siguientes. Las instrucciones etiquetadas no son requisitos sintácticos, pero la `switch` instrucción no tiene sentido sin ellas. Dos `constant-expression` valores de `case` las instrucciones no pueden evaluarse como el mismo valor. La `default` etiqueta solo puede aparecer una vez. `default` A menudo, la instrucción se coloca al final, pero puede aparecer en cualquier parte del cuerpo de la `switch` instrucción. Una etiqueta `case` o `default` solo puede aparecer en una instrucción `switch`.

`constant-expression` En cada `case` etiqueta se convierte en un valor constante que es del mismo tipo que `condition`. A continuación, se compara con `condition` para determinar si son iguales. El control pasa a la primera instrucción después del `case` `constant-expression` valor que coincide con el valor de `condition`. El comportamiento resultante se muestra en la siguiente tabla.

switch comportamiento de la instrucción

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.

CONDICIÓN	ACCIÓN
Ninguna de las constantes coincide con las constantes en las <code>case</code> etiquetas; existe una <code>default</code> etiqueta.	El control se transfiere a la <code>default</code> etiqueta.
Ninguna de las constantes coincide con las constantes en las <code>case</code> etiquetas; no <code>default</code> existe ninguna etiqueta.	El control se transfiere a la instrucción después de la <code>switch</code> instrucción.

Si se encuentra una expresión coincidente, la ejecución puede continuar a través `case` de las etiquetas posteriores o `default`. La `break` instrucción se utiliza para detener la ejecución y transferir el control a la instrucción después de la `switch` instrucción. Sin una `break` instrucción, se ejecuta cada instrucción de la `case` etiqueta coincidente hasta el final de `switch`, incluido `default`. Por ejemplo:

```
// 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 `case 'A'`. La `break` instrucción después `uppercase_A++` de finaliza la ejecución del `switch` cuerpo de la instrucción y el control pasa al `while` bucle. Sin la `break` instrucción, la ejecución pasaría a la siguiente instrucción etiquetada, por lo que `lowercase_a` `other` también se incrementaría. La instrucción de proporciona un propósito similar `break` `case 'a'`. Si `c` es menor `case 'a'`, `lowercase_a` se incrementa y la `break` instrucción finaliza el cuerpo de la `switch` instrucción. Si `c` no es `'a'` ni `'A'`, `default` se ejecuta la instrucción.

Visual Studio 2017 y versiones posteriores: (disponible con [/STD:c++17](#)) el `[[fallthrough]]` atributo se especifica en el estándar c++ 17. Se puede utilizar en una `switch` instrucción. Es una sugerencia para el compilador, o cualquier persona que lea el código, el comportamiento de paso a través es intencionado. El compilador de Microsoft C++ no advierte actualmente sobre el comportamiento de fallthrough, por lo que este atributo no tiene ningún efecto en el comportamiento del compilador. En el ejemplo, el atributo se aplica a una instrucción vacía dentro de la instrucción etiquetada sin terminar. Es decir, es necesario el punto y coma.

```

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 posterior (disponible con [/STD: c++ 17](#)). Una `switch` instrucción puede tener una `init-statement` cláusula, que finaliza con un punto y coma. Introduce e inicializa una variable cuyo ámbito se limita al bloque de la `switch` instrucción:

```

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 `switch` instrucción puede contener definiciones con inicializadores siempre que sean *accesibles*, es decir, no omitir todas las rutas de acceso de ejecución posibles. Los nombres proporcionados mediante estas declaraciones tienen ámbito local. Por ejemplo:

```
// 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 `switch` instrucción puede estar anidada. Cuando está anidada, `case` las `default` etiquetas o se asocian con la instrucción más cercana `switch` que las incluye.

Comportamiento específico de Microsoft

Microsoft C++ no limita el número de `case` valores en una `switch` instrucción. El número solo está limitado por la memoria disponible.

Consulte también

[Instrucciones de selección](#)

[Palabras clave](#)

Instrucciones de iteración (C++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 [for basada en intervalo](#)). Cada una de estas iteraciones hasta que su expresión de finalización se evalúa como cero (false) o hasta que la terminación del bucle se fuerza con una `break` instrucción. 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.

Consulta también

[Información general sobre las instrucciones de C++](#)

while (Instrucción) (C++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

Ejecuta la *instrucción* repetidamente hasta que la *expresión* se evalúa como cero.

Sintaxis

```
while ( expression )
    statement
```

Observaciones

La prueba de *Expression* tiene lugar antes de cada ejecución del bucle; por lo tanto, un `while` bucle se ejecuta cero o más veces. La *expresión* debe ser de un tipo entero, un tipo de puntero o un tipo de clase con una conversión no ambigua a un tipo entero o de puntero.

Un `while` bucle también puede finalizar cuando se ejecuta `break`, `goto` o `Return` dentro del cuerpo de la instrucción. Use `continuar` para finalizar la iteración actual sin salir del `while` bucle. `continue` pasa el control a la siguiente iteración del `while` bucle.

En el código siguiente se usa un `while` bucle para recortar los guiones bajos finales de una cadena:

```
// 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.

Consulta también

[Instrucciones de iteración](#)

[Palabras clave](#)

do-while (Instrucción) (C++)
for (instrucción) (C++)
Instrucción for basada en intervalo (C++)

do-while (instrucción de C++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

Ejecuta una *instrucción* varias veces hasta que la condición de finalización especificada (la *expresión*) se evalúa como cero.

Sintaxis

```
do
    statement
  while ( expression ) ;
```

Observaciones

La prueba de la condición de finalización se realiza después de cada ejecución del bucle. por lo tanto, 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 ejemplo siguiente se muestra la instrucción **do-while** :

```
// do_while_statement.cpp
#include <stdio.h>
int main()
{
    int i = 0;
    do
    {
        printf_s("\n%d",i++);
    } while (i < 3);
}
```

Consulta también

[Instrucciones de iteración](#)

[Palabras clave](#)

[while \(Instrucción\) \(C++\)](#)

[for \(instrucción\) \(C++\)](#)

[Instrucción for basada en intervalo \(C++\)](#)

for Statement (C++)

02/11/2020 • 6 minutes to read • [Edit Online](#)

Ejecuta una instrucción repetidamente hasta que la condición es false. Para obtener información sobre la instrucción basada en intervalo `for`, vea [instrucción basada en intervalo `for` \(C++\)](#).

Syntax

```
for ( init-expression ; cond-expression ; Loop-expression )  
    statement
```

Comentarios

Use la `for` instrucción para construir bucles que deben ejecutar un número especificado de veces.

La `for` instrucción consta de tres partes opcionales, tal como se muestra en la tabla siguiente.

elementos de bucle for

NOMBRE DE SINTAXIS	CUANDO SE EJECUTA	DESCRIPCIÓN
<code>init-expression</code>	Antes de cualquier otro elemento de la <code>for</code> instrucción, <code>init-expression</code> se ejecuta una sola vez. A continuación, el control pasa 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 se <code>cond-expression</code> 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.
<code>Loop-expression</code>	Al final de cada iteración de <code>statement</code> . Una vez que <code>Loop-expression</code> se ejecuta, <code>cond-expression</code> se evalúa.	Se utiliza normalmente para incrementar índices de bucle.

En los siguientes ejemplos se muestran distintas formas de usar la `for` instrucción.

```

#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:

```

#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 incrementar o reducir, o modificar de otras maneras.

```

#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 `for` bucle finaliza cuando `break` se ejecuta, `Retorno goto` (en una instrucción con etiqueta fuera del `for` bucle) dentro de `statement`. Una `continue` instrucción de un `for` bucle finaliza solo la iteración actual.

Si `cond-expression` se omite, se considera `true` y el `for` bucle no finalizará sin un `break`, `return` o `goto` dentro de `statement`.

Aunque los tres campos de la `for` instrucción se utilizan normalmente para la inicialización, las pruebas de

finalización y el incremento, no están restringidos a estos usos. Por ejemplo, el código siguiente imprime los números de 0 a 4. En este caso, `statement` es la instrucción NULL:

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

int main()
{
    int i;
    for( i = 0; i < 5; cout << i << '\n', i++){
        ;
    }
}
```

for bucles y el estándar de C++

El estándar de C++ indica que una variable declarada en un `for` bucle debe salir del ámbito después de que `for` finalice el bucle. Por ejemplo:

```
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 `for` bucle permanece dentro del ámbito hasta que `for` finaliza el ámbito de inclusión del bucle.

`/Zc:forScope` permite el comportamiento estándar de las variables declaradas en bucles for sin necesidad de especificar `/Za`.

También es posible usar las diferencias de ámbito del `for` bucle para volver a declarar las variables de la `/ze` siguiente manera:

```
// 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++ ) {}
}
```

Este comportamiento imita de forma más estrecha el comportamiento estándar de una variable declarada en un `for` bucle, que requiere que las variables declaradas en un bucle salgan `for` del ámbito una vez terminado el bucle. Cuando una variable se declara en un `for` bucle, el compilador la promueve internamente a una variable local en el `for` ámbito de inclusión del bucle. Se promueve incluso si ya existe una variable local con el mismo nombre.

Vea también

[Instrucciones de iteración](#)

[Palabras clave](#)

[while \(instrucción\) \(C++\)](#)

[do-while \(instrucción\) \(C++\)](#)

[Instrucción for basada en intervalo \(C++\)](#)

Instrucción for basada en intervalo (C++)

06/03/2021 • 4 minutes to read • [Edit Online](#)

Ejecuta `statement` de forma repetida y secuencial para cada elemento de `expression`.

Sintaxis

```
* for ( ***for-Range-declaration : expresión de )  
    instrucción
```

Observaciones

Use la instrucción basada en intervalo `for` para construir bucles que se deben ejecutar a través de un *intervalo*, que se define como cualquier elemento que se pueda 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 `for-range-declaration` parte es local a la `for` instrucción y no se puede volver a declarar en `expression` o `statement`. Tenga en cuenta que la `auto` palabra clave es preferible en la `for-range-declaration` parte de la instrucción.

Novedades de Visual Studio 2017: Los bucles basados en intervalos ya `for` no requieren que `begin()` y `end()` devuelvan objetos del mismo tipo. Esto permite `end()` que devuelva un objeto centinela, como lo usan los intervalos según se define en la propuesta Ranges-V3. Para obtener más información, vea [generalizar el For bucle de Range-Based](#) y la [biblioteca Range-V3 en github](#).

Este código muestra cómo usar bucles basados en intervalos `for` para recorrer en iteración una matriz y un vector:

```

// 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:

```

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 basado en intervalo `for` finaliza cuando se ejecuta uno de estos en `statement` : a `break` , `return` o `goto` a una instrucción con etiqueta fuera del bucle basado en intervalo `for` . Una `continue` instrucción de un bucle basado en intervalo `for` finaliza solo la iteración actual.

Tenga en cuenta que estos datos se basan en el intervalo `for` :

- 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.

Consulta también

[auto](#)

[Instrucciones de iteración](#)

[Palabras clave](#)

[Instrucción while \(C++\)](#)

[Instrucción do-while \(C++\)](#)

[Instrucción for \(C++\)](#)

Instrucciones de salto (C++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

Una instrucción de salto de C++ realiza una transferencia de control local inmediata.

Sintaxis

```
break;  
continue;  
return [expression];  
goto identifier;
```

Observaciones

Vea los temas siguientes para obtener una descripción de las instrucciones de salto de C++.

- [break \(Instrucción\)](#)
- [continue \(Instrucción\)](#)
- [return \(instrucción\)](#)
- [goto \(instrucción\)](#)

Consulta también

[Información general sobre las instrucciones de C++](#)

break (Instrucción) (C++)

06/03/2021 • 3 minutes to read • [Edit Online](#)

La `break` instrucción finaliza la ejecución del bucle envolvente o la instrucción condicional más cercanos en los que aparece. El control pasa a la instrucción que hay a continuación del final de la instrucción, si hay alguna.

Sintaxis

```
break;
```

Observaciones

La `break` instrucción se usa con la instrucción `Switch` condicional y con las instrucciones de bucle `do`, `for` y `While`.

En una `switch` instrucción, la `break` instrucción hace que el programa ejecute la siguiente instrucción fuera de la `switch` instrucción. Sin una `break` instrucción, se ejecuta cada instrucción de la `case` etiqueta coincidente hasta el final de la `switch` instrucción, incluida la `default` cláusula.

En los bucles, la `break` instrucción finaliza la ejecución de la `do` instrucción envolvente, o más cercana `for` o `while`. El control pasa a la instrucción que hay a continuación de la instrucción finalizada, si hay alguna.

Dentro de las instrucciones anidadas, la `break` instrucción finaliza solo la `do` instrucción, `for`, `switch` o `while` que la incluye inmediatamente. Puede usar una `return` instrucción o `goto` para transferir el control desde estructuras anidadas más profundamente.

Ejemplo

En el código siguiente se muestra cómo usar la `break` instrucción en un `for` bucle.

```

#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';
    }
}

```

In each case:

1
2
3

En el código siguiente se muestra cómo utilizar `break` en un `while` bucle y un `do` bucle.

```

#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);
}

```

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 controlar cada caso por separado; si no usa `break`, la ejecución del código pasa al siguiente caso.

```
#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";
    }
}
```

Consulta también

[Instrucciones de salto](#)

[Palabras clave](#)

[continue \(Instrucción\)](#)

continue (Instrucción) (C++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

Fuerza la transferencia de control a la expresión de control del bucle [de inclusión más pequeño](#), [foro While](#).

Sintaxis

```
continue;
```

Observaciones

No se ejecuta ninguna de las instrucciones restantes de la iteración actual. La siguiente iteración del bucle se determina del modo siguiente:

- En un `do` `while` bucle o, la siguiente iteración comienza reevaluando la expresión de control de la `do` `while` instrucción o.
- En un `for` bucle (con la sintaxis `for(<init-expr> ; <cond-expr> ; <loop-expr>)`), `<loop-expr>` se ejecuta la cláusula. 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 `continue` se puede usar la instrucción para omitir secciones de código e iniciar la siguiente iteración de un bucle.

Ejemplo

```
// continue_statement.cpp
#include <stdio.h>
int main()
{
    int i = 0;
    do
    {
        i++;
        printf_s("before the continue\n");
        continue;
        printf("after the continue, should never print\n");
    } while (i < 3);

    printf_s("after the do loop\n");
}
```

```
before the continue
before the continue
before the continue
after the do loop
```

Consulte también

[Instrucciones de salto](#)

[Palabras clave](#)

return (Instrucción) (C++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

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];
```

Observaciones

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 `return` tipo de la función puede crear objetos temporales. Para obtener más información sobre cómo y cuándo se crean objetos temporales, vea [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, y las funciones de tipo `void`, no pueden especificar una expresión en la `return` instrucción. Las funciones de todos los demás tipos deben especificar una expresión en la `return` instrucción.

Cuando el flujo de control sale del bloque que contiene la definición de función, el resultado es el mismo que si se `return` hubiera ejecutado una instrucción 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 `return` instrucciones.

En el ejemplo siguiente se usa una expresión con una `return` instrucción para obtener el mayor de dos enteros.

Ejemplo

```
// 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++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

La `goto` instrucción transfiere el control incondicionalmente a la instrucción etiquetada por el identificador especificado.

Sintaxis

```
goto identifier;
```

Observaciones

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 `goto` instrucción; de lo contrario, se omiten las etiquetas de instrucción. Las etiquetas no se pueden volver a declarar.

Una `goto` instrucción no puede transferir 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 genera C2362:

```
int goto_fn(bool b)
{
    if (!b)
    {
        goto exit; // C2362
    }
    else
    { /*...*/ }

    int error_code = 42;

exit:
    return error_code;
}
```

Es un buen estilo de programación utilizar las `break`, `continue` instrucciones, y `return` en lugar de la `goto` instrucción siempre que sea posible. Sin embargo, dado que la `break` instrucción sale solo de un nivel de un bucle, puede que tenga que usar una `goto` instrucción para salir de un bucle profundamente anidado.

Para obtener más información sobre las etiquetas y la `goto` instrucción, vea [instrucciones con etiqueta](#).

Ejemplo

En este ejemplo, una `goto` instrucción transfiere el control al punto con la etiqueta `stop` cuando `i` es igual a 3.

```

// 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 );
}

```

```

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

```

Consulta también

[Instrucciones de salto](#)

[Palabras clave](#)

Transferencias del control

06/03/2021 • 2 minutes to read • [Edit Online](#)

Puede usar la `goto` instrucción o una `case` etiqueta en una `switch` instrucción para especificar un programa que se bifurca después 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`. También hay una instrucción errónea `goto` que transfiere el control más allá de un inicializador.

```
// 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 `goto` instrucción 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`, que se declaran en el bloque que actúa como *instrucción* de la `while` instrucción, se destruyen cuando se sale del bloque mediante la `break` instrucción.

Espacios de nombres (C++)

06/03/2021 • 14 minutes to read • [Edit Online](#)

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 mediante 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.

```
namespace ContosoData
{
    class ObjectManager
    {
        public:
            void DoSomething() {}
    };
    void Func(ObjectManager) {}
}
```

Use el nombre completo:

```
ContosoData::ObjectManager mgr;
mgr.DoSomething();
ContosoData::Func(mgr);
```

Use una declaración using para poner un identificador en el ámbito:

```
using ContosoData::ObjectManager;
ObjectManager mgr;
mgr.DoSomething();
```

Use una directiva using para poner todo el espacio de nombres en el ámbito:

```
using namespace ContosoData;

ObjectManager mgr;
mgr.DoSomething();
Func(mgr);
```

directivas Using

La `using` directiva permite que todos los nombres de `namespace` se utilicen sin el *nombre de espacio de nombres* como calificador explícito. Use una directiva using en un archivo de implementación (es decir, *.cpp) si usa varios identificadores diferentes en un espacio de nombres; Si solo usa uno o dos identificadores, considere

la posibilidad de usar una declaración Using para traer 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.

NOTE

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.

```
//contosoData.h
#pragma once
namespace ContosoDataServer
{
    void Foo();
    int Bar();
}
```

Las implementaciones de funciones en contosodata.cpp deben usar el nombre completo, incluso si se coloca una `using` Directiva en la parte superior del archivo:

```
#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:

```
// 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 realizar declaraciones en el ámbito global siempre que sea posible, salvo la [función principal](#) de punto de entrada, que debe 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

Todos los tipos y funciones de la biblioteca estándar de C++ se declaran en el espacio de nombres `std` o 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 sin calificar 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 inline), como se muestra en el ejemplo siguiente:

```
namespace ContosoDataServer
{
    void Foo();

    namespace Details
    {
        int CountImpl;
        void Bar() { return Foo(); }
    }

    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 anidado. También permite declarar una especialización en un espacio de nombres primario para una plantilla que se declara en el espacio de nombres anidado. En el ejemplo siguiente se muestra cómo el código externo enlaza de forma predeterminada con el espacio de nombres anidado:

```
//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 anidado:

```
namespace Parent
{
    inline namespace new_ns
    {
        template <typename T>
        struct C
        {
            T member;
        };
    }
    template<>
    class C<int> {};
}
```

Puede usar espacios de nombres anidados 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 anidado 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`.

```
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 espacio 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 es tedioso escribirlo en un archivo de encabezado donde no se pueden usar las directivas `Using`, puede crear un alias de espacio de nombres que sirva como una abreviatura para el nombre real. Por ejemplo:

```
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:

```
namespace
{
    int MyFunc(){}
}
```

Esto se denomina espacio de nombres sin nombre o anónimo y es útil cuando se desea que las declaraciones de variables no sean visibles para el código de otros archivos (es decir, se les proporciona una vinculación interna) sin tener que crear un espacio de nombres con nombre. 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.

Consulta también

[Declaraciones y definiciones](#)

Enumeraciones [C++]

06/03/2021 • 9 minutes to read • [Edit Online](#)

Una enumeración es un tipo definido por el usuario que consta de un conjunto de constantes enteras con nombre conocidas como enumeradores.

NOTE

En este artículo se trata el tipo de lenguaje C++ estándar ISO `enum` y el tipo de **clase de enumeración** con ámbito (o fuertemente tipado) que se introduce en C++ 11. Para obtener información sobre la **clase de enumeración pública** o los tipos de **clase de enumeración privada** en c++/CLI y c++/CX, vea [enum \(clase\)](#).

Sintaxis

```
// 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.

enumeración de lista

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 circundante; en una enumeración con ámbito, el ámbito es la *lista de enumeración* propiamente dicha. En una enumeración con ámbito, la lista puede estar vacía que, en efecto, define un nuevo tipo entero.

class

Mediante el uso de esta palabra clave en la declaración, se especifica que la enumeración tiene el ámbito y se debe proporcionar un *identificador*. También puede usar la `struct` palabra clave en lugar de `class`, ya que son semánticamente equivalentes en este contexto.

Ámbito del enumerador

Una enumeración proporciona contexto para describir un intervalo de valores que se representan como constantes con nombre, y también se denominan enumeradores. En los tipos de enumeración original 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:

```
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:

```
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 en el que `enum` se define (para las enumeraciones sin ámbito) o en el `enum` propio (para las enumeraciones de ámbito). Los valores especificados en los nombres no tienen que ser únicos. Por ejemplo, si la declaración de una enumeración sin ámbito `Suit` es esta:

```
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 utiliza 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 se puede convertir implícitamente a un valor de enumeración. El ejemplo siguiente muestra lo que ocurre si

intenta asignar a `hand` un valor que no sea `Suit`:

```
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` en un enumerador de ámbito o sin ámbito. Sin embargo, puede promover un enumerador sin ámbito a un valor entero sin una conversión.

```
int account_num = Hearts; //OK if Hearts is in a unscoped 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 enumeración (identificador) y no pueden convertirse implícitamente, como se muestra en el ejemplo siguiente:

```
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 posterior (disponible con [/STD: c++ 17](#)): al definir una enumeración (normal o con ámbito) con un tipo subyacente explícito y sin enumeradores, puede aplicar un nuevo tipo entero que no tiene ninguna conversión implícita a ningún otro tipo. Al usar este tipo en lugar de su tipo subyacente integrado, puede eliminar la posibilidad de que se produzcan errores sutiles causados por conversiones implícitas involuntarias.

```
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 Abi sin penalización del rendimiento. No se requiere ninguna conversión cuando las variables del tipo se inicializan mediante la inicialización de lista directa. En el ejemplo siguiente se muestra cómo inicializar enumeraciones sin enumeradores en varios contextos:

```
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;
}
```

Consulta también

[Declaraciones de enumeración de C](#)

[Palabras clave](#)

NOTE

En C++ 17 y versiones posteriores, `std::variant` class es una alternativa con seguridad de tipos para 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, un union no puede contener más de un objeto de su lista de miembros. También significa que, independientemente del número de miembros que union tenga, siempre usa la memoria suficiente para almacenar el miembro más grande.

unionPuede ser útil para la reserva de memoria cuando se tienen muchos objetos y memoria limitada. Sin embargo, union requiere atención adicional para utilizarlo correctamente. Usted es responsable de asegurarse de que siempre tiene acceso al mismo miembro que ha asignado. Si algún tipo de miembro tiene un con o no trivial struct , debe escribir código adicional para tener struct y destruir explícitamente ese miembro. Antes de usar un union , considere si el problema que está intentando resolver podría expresarse mejor mediante el uso de class tipos base y derivados class .

Sintaxis

```
union *** tag * opt { OPT * member-list *** };
```

Parámetros

`tag`

Nombre del tipo dado al union .

`member-list`

Miembros que union puede contener.

Declarar un union

Comience la declaración de un union mediante la `union` palabra clave y escriba la lista de miembros entre llaves:

```

// 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 un union

En el ejemplo anterior, cualquier código que tenga acceso a union necesita saber qué miembro contiene los datos. La solución más común a este problema se denomina **discriminada union**. Agrega el elemento union en un struct e incluye un enum miembro que indica el tipo de miembro almacenado actualmente en union . En el ejemplo siguiente se muestra el patrón básico:

```

#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, union en `Input` struct no tiene nombre, por lo que se denomina *anónimo* union . Se puede tener acceso a sus miembros directamente como si fueran miembros de struct . Para obtener más información sobre cómo usar un anónimo union , consulte la [sección union anónima](#) .

En el ejemplo anterior se muestra un problema que también se podría resolver mediante el uso class de tipos que derivan de una base común class . Puede crear una bifurcación del código en función del tipo en tiempo de ejecución de cada objeto del contenedor. Es posible que el código sea más fácil de mantener y comprender, pero también puede ser más lento que el uso de union . Además, con union , puede almacenar tipos no relacionados. unionPermite cambiar dinámicamente el tipo del valor almacenado sin cambiar el tipo de la union propia variable. Por ejemplo, podría crear una matriz heterogénea de `MyUnionType` , cuyos elementos almacenan valores diferentes de tipos diferentes.

Es fácil utilizar el uso incorrecto del `Input` struct en el ejemplo. Es el usuario quien debe usar el discriminador correctamente para tener acceso al miembro que contiene los datos. Puede protegerse contra el uso indebido si hace que union `private` y proporcione funciones de acceso especiales, como se muestra en el ejemplo siguiente.

Unrestricted union (c++ 11)

En C++ 03 y versiones anteriores, un union puede contener miembros que no son de static datos que tienen un class tipo, siempre que el tipo no tenga ningún usuario proporcionado con los struct operadores Ors, de struct Ors o de asignación. En C++11, se quitaron estas restricciones. Si incluye un miembro de este tipo en union , el compilador marca automáticamente las funciones miembro especiales que no proporciona el usuario como

`deleted` . Si union es un elemento anónimo union dentro de class o struct , todas las funciones miembro especiales de class o struct que no son proporcionadas por el usuario se marcan como `deleted` . En el ejemplo siguiente se muestra cómo controlar este caso. Uno de los miembros de union tiene un miembro que requiere este tratamiento especial:

```
// 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:
                break;
        }
    }
};
```

```

        _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;
}

```

```

        _ASSERT(this->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->kind_) MyVariant(Kind::A);
        this->a_ = a;
    }
    return *this;
}

```

```

        new (this) MyVariant(d);
    }
    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;
}

```

Un union no puede almacenar una referencia. unionTampoco admite la herencia. Esto significa que no puede usar un union como base class o heredar de otro class o tener funciones virtuales.

Inicializa un union

Puede declarar e inicializar union en la misma instrucción asignando una expresión entre llaves. La expresión se evalúa y se asigna al primer campo de union .

```

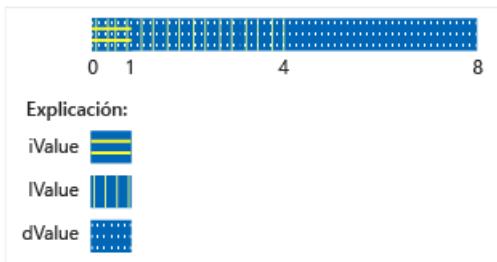
#include <iostream>
using namespace std;

union NumericType
{
    short      iValue;
    long       lValue;
    double     dValue;
};

int main()
{
    union NumericType Values = { 10 }; // iValue = 10
    cout << Values.iValue << endl;
    Values.dValue = 3.1416;
    cout << Values.dValue << endl;
}
/* Output:
10
3.141600
*/

```

`NumericType` union Se organiza en memoria (conceptualmente) como se muestra en la ilustración siguiente.



Almacenamiento de datos en un `NumericType` Union

Anonymous union

Un anónimo union es uno declarado sin un `class-name` o `declarator-list`.

```
union { member-list }
```

Los nombres declarados en un anónimo union se utilizan directamente, como las variables no miembro. Implica que los nombres declarados en un anónimo union deben ser únicos en el ámbito circundante.

Un anónimo union está sujeto a estas restricciones adicionales:

- Si se declara en el ámbito de archivo o espacio de nombres, también se debe declarar como `static`.
- Solo puede tener `public` miembros; tener `private` y `protected` los miembros de un anónimo union genera errores.
- No puede tener funciones miembro.

Consulte también

[Clases y structs](#)

[Palabras clave](#)

`class`

`struct`

Funciones (C++)

06/03/2021 • 24 minutes to read • [Edit Online](#)

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 función siguiente acepta dos enteros de un llamador y devuelve su suma; *a* y *b* son *parámetros* de tipo `int`.

```
int sum(int a, int b)
{
    return a + b;
}
```

Se puede invocar la función o *llamarla* desde cualquier número de lugares del programa. Los valores que se pasan a la función son los *argumentos*, cuyos tipos deben ser compatibles con los tipos de parámetro de la definición de función.

```
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 un buen diseño tiene como objetivo 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 en gran medida en la biblioteca estándar.

Las funciones se pueden *sobrecargar*, lo que significa que las distintas versiones de una función pueden compartir el mismo nombre si difieren en el número o el tipo de parámetros formales. Para obtener más información, vea [sobrecarga de funciones](#).

Elementos de una declaración de función

Una *declaración* de función mínima está formada por el tipo de valor devuelto, el nombre de función y la lista de parámetros (que puede estar vacía), junto con palabras clave opcionales que proporcionan instrucciones adicionales para el compilador. El ejemplo siguiente es una declaración de función:

```
int sum(int a, int b);
```

Una definición de función se compone de una declaración, más el *cuerpo*, que es todo el código entre las llaves:

```
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 valor devuelto, 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 valor devuelto válido que indica al compilador que infiera el tipo a partir de la instrucción `return`. En C++ 14, `decltype(auto)` también se permite. Para obtener más información, consulte más adelante Deducción de tipos en tipos de valor devueltos.
2. El nombre de función, que debe comenzar con una letra o un carácter de subrayado y no puede contener espacios. En general, un carácter de subrayado inicial en los nombres de función de la biblioteca estándar indica funciones de miembro privado o funciones no miembro que no están pensadas para que las use el 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.

```
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`.

```
//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 por 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.

```
inline double Account::GetBalance()
{
    return balance;
}
```

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

4. Una `noexcept` expresión, que especifica si la función puede producir o no una excepción. En el ejemplo siguiente, la función no produce una excepción si la `is_pod` expresión se evalúa como `true`.

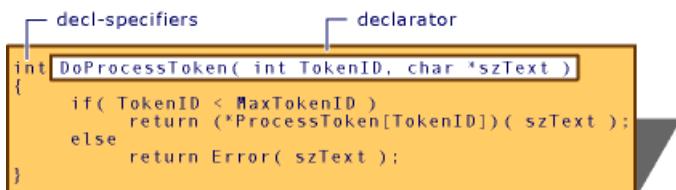
```
#include <type_traits>

template <typename T>
T copy_object(T& obj) noexcept(std::is_pod<T>) {...}
```

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

5. (Solo funciones miembro) Los calificadores CV, que especifican si la función es `const` o `volatile`.
6. (Solo funciones miembro) `virtual`, `override` o `final`. `virtual` Especifica que una función se puede invalidar 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 obtener más información, vea [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 la sobrecarga de una función que se va a elegir cuando el parámetro de objeto implícito (`*this`) es una referencia de valor r en lugar de una referencia lvalue. Para obtener más información, vea [sobrecarga de funciones](#).

La ilustración siguiente muestra las partes de una definición de función. El área sombreada es el cuerpo de la función.



Elementos de una definición de función

Definiciones de función

Una *definición de función* está formada por la declaración y el cuerpo de la función, entre llaves, que contiene declaraciones de variables, instrucciones y expresiones. En el ejemplo siguiente se muestra una definición de función completa:

```
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.

```
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 ningún miembro de datos de la clase. Al declarar una función miembro como `const`, ayuda al compilador a aplicar la *corrección const*. Si alguien intenta modificar incorrectamente el objeto con una función declarada como `const`, se genera un error del compilador. Para obtener más información, vea [const](#).

Declare una función como `constexpr` cuando el valor que produce puede determinarse en tiempo de compilación. Una función `constexpr` normalmente se ejecuta más rápido que una función normal. 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.

```
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, vea [plantillas de función](#).

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 parámetros adicionales de tipo de 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 los objetos grandes, puede resultar costoso realizar una copia y no siempre es necesario. Para hacer que los argumentos se pasen por referencia (concretamente, referencia de valor `&`), agregue un cuantificador de referencia al parámetro:

```
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 este tipo de argumento, califique el parámetro como `const&`:

```
void DoSomething(const std::string& input){...}
```

C++ 11: Para controlar explícitamente los argumentos que se pasan por referencia rvalue o lvalue-Reference, use un signo de y comercial en el parámetro para indicar una referencia universal:

```
void DoSomething(const std::string&& input){...}
```

Una función declarada con la palabra clave Single `void` en la lista de declaraciones de parámetros no toma ningún argumento, siempre y cuando la palabra clave `void` sea el primer y único miembro de la lista de declaraciones de argumentos. Los argumentos de tipo `void` en otro lugar de la lista producen errores. Por ejemplo:

```
// OK same as GetTickCount()
long GetTickCount( void );
```

Tenga en cuenta que, 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 parte 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.

```
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, vea [argumentos predeterminados](#).

Tipos de valor devuelto de función

Una función no puede devolver otra función o una matriz integrada; sin embargo, puede devolver punteros a estos tipos, o una *expresión lambda*, que genera un objeto de función. Excepto en estos casos, una función puede devolver un valor de cualquier tipo que esté en el ámbito, o bien no puede devolver ningún valor, en cuyo caso el tipo de valor devuelto 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 está 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.

```
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, solo sirve como marcador de posición para cualquier cosa que genere la expresión `decltype` y no realiza la deducción de tipos.

Variables locales de función

Una variable que se declara dentro de un cuerpo de función se denomina *variable local* o simplemente *local*. Las variables locales no estáticas solo son visibles dentro del cuerpo de función y, si se declaran en la pila, se salen del ámbito cuando finaliza la función. Cuando se crea una variable local y se devuelve por valor, el compilador normalmente puede realizar la *optimización del valor devuelto 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 crea un objeto estático porque el flujo de control de programa omitió su declaración, no se realiza ningún intento de destruir ese objeto.

Deducción de tipos en tipos de valor devueltos (C++ 14)

En C++ 14, puede usar `auto` para indicar al compilador que infiera el tipo de valor devuelto desde el cuerpo de la función sin tener que proporcionar un tipo de valor devuelto final. Tenga en cuenta que `auto` siempre se deduce como devolución por valor. Use `auto&&` para indicar al compilador que deduzca una referencia.

En este ejemplo, se `auto` deducirá como una copia de valor no const de la suma de LHS y RHS.

```
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 no `auto` conserva la constante del tipo que se deduce. En el caso de las funciones de reenvío cuyo valor devuelto tiene que conservar la constante o la referencia de sus argumentos, puede usar la `decltype(auto)` palabra clave, que utiliza las `decltype` reglas de inferencia de tipos y conserva toda la información de tipo. `decltype(auto)` se puede utilizar como valor devuelto ordinario en el lado izquierdo o como un valor devuelto final.

En el ejemplo siguiente (basado en código de [N3493](#)), `decltype(auto)` se muestra que se usa para habilitar el reenvío directo de los argumentos de función en un tipo de valor devuelto que no se conoce hasta que se crea una instancia de la plantilla.

```

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());
}

```

Devolver varios valores de una función

Hay varias maneras de devolver más de un valor de una función:

1. Encapsula los valores en un objeto de clase o struct con nombre. Requiere que la definición de la clase o el struct sea visible para el autor de la llamada:

```

#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::p Air:

```

#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 con [/std:c+17](#)): Use 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 mucho más eficaz. En la instrucción `auto[x, y, z] = f();`, los corchetes introducen e inicializan los nombres que están en el ámbito de todo el bloque de función.

```

#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 usar el propio valor devuelto, puede "devolver" los valores definiendo cualquier número de parámetros para usar pass-by-Reference para que la función pueda modificar o inicializar los valores de los objetos que proporciona el autor de la llamada. Para obtener más información, vea [argumentos de función de tipo de 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 `typedef` usar para declarar un alias para el tipo de puntero de función si se declara una función que devuelve un tipo de puntero de función. Por ejemplo

```

typedef int (*fp)(int);
fp myFunction(char* s); // function returning function pointer

```

Si no es así, la sintaxis correcta para la declaración de la función se puede deducir de la sintaxis de declarador del puntero a función, mediante la sustitución del identificador (`fp` en el ejemplo anterior) por el nombre y la lista de argumentos de las funciones, como sigue:

```

int (*myFunction(char* s))(int);

```

La declaración anterior es equivalente a la declaración anterior `typedef`.

Consulta también

[Sobrecarga de funciones](#)

[Funciones con listas de argumentos de variable](#)

[Funciones predeterminadas y eliminadas explícitamente](#)

[Búsqueda de nombres dependientes de argumentos \(Koenig\) en funciones](#)

[Argumentos predeterminados](#)

[Funciones insertadas](#)

Funciones con listas de argumentos de variable (C++)

06/03/2021 • 5 minutes to read • [Edit Online](#)

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 utilizan listas de argumentos variables. `printf` *lista de declaraciones de argumentos*

Funciones con argumentos variables

Para obtener acceso a los argumentos después de los declarados, 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`. Del mismo modo, 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. Vea [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 <stdarg.h> archivo de inclusión para tener acceso a los argumentos que pasa una lista de variables. Para obtener más información sobre estas macros, vea [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 junto con el tipo (declarado en <stdarg.h>):

```
// variable_argument_lists.cpp
#include <stdio.h>
#include <stdarg.h>

// Declaration, but not definition, of ShowVar.
void ShowVar( char *szTypes, ... );
int main() {
    ShowVar( "fcsi", 32.4f, 'a', "Test string", 4 );
}

// ShowVar takes a format string of the form
// "ifcs", where each character specifies the
// type of the argument in that position.
//
```

```

// i = int
// f = float
// c = char
// s = string (char *)
//
// Following the format specification is a variable
// list of arguments. Each argument corresponds to
// a format character in the format string to which
// the szTypes parameter points
void ShowVar( char *szTypes, ... ) {
    va_list vl;
    int i;

    // szTypes is the last argument specified; you must access
    // all others using the variable-argument macros.
    va_start( vl, szTypes );

    // Step through the list.
    for( i = 0; szTypes[i] != '\0'; ++i ) {
        union Printable_t {
            int     i;
            float   f;
            char    c;
            char   *s;
        } Printable;

        switch( szTypes[i] ) {    // Type to expect.
            case 'i':
                Printable.i = va_arg( vl, int );
                printf_s( "%i\n", Printable.i );
                break;

            case 'f':
                Printable.f = va_arg( vl, double );
                printf_s( "%f\n", Printable.f );
                break;

            case 'c':
                Printable.c = va_arg( vl, char );
                printf_s( "%c\n", Printable.c );
                break;

            case 's':
                Printable.s = va_arg( vl, char * );
                printf_s( "%s\n", Printable.s );
                break;

            default:
                break;
        }
    }
    va_end( vl );
}

//Output:
// 32.400002
// a
// Test string

```

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 desee.

Debe llamar a la macro para finalizar el procesamiento del argumento de variable. `va_end`

Sobrecarga de funciones

06/03/2021 • 34 minutes to read • [Edit Online](#)

C++ permite especificar más de una función del mismo nombre en el mismo ámbito. Estas funciones se denominan funciones *sobrecargadas*. Las funciones sobrecargadas permiten proporcionar una semántica diferente para una función, en función de los tipos y el número de argumentos.

Por ejemplo, una `print` función que toma un `std::string` argumento podría realizar tareas muy diferentes a las que toma un argumento de tipo `double`. La sobrecarga evita tener que usar nombres como `print_string` o `print_double`. En tiempo de compilación, el compilador elige qué sobrecarga se debe usar en función del tipo de argumentos pasado por el llamador. Si llama a `print(42.0)`, se `void print(double d)` invocará la función. Si llama a `print("hello world")`, se `void print(std::string)` invocará la sobrecarga.

Puede sobrecargar las funciones miembro y las funciones no miembro. 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 <code>typedef</code> nombres	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	Sí

Ejemplo

En el ejemplo siguiente se muestra cómo se puede usar la sobrecarga.

```
// 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.
```

```

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 utiliza en la selección de funciones sobrecargadas. Dos funciones que solo difieren en sus argumentos predeterminados se consideran varias definiciones en lugar de funciones sobrecargadas.

No se pueden proporcionar argumentos predeterminados para operadores sobrecargados.

Coincidencia de argumentos

Las funciones sobrecargadas se seleccionan para obtener la mejor coincidencia entre las declaraciones de función del ámbito y los argumentos proporcionados en la llamada de función. Si se encuentra una función adecuada, se llama a esa función. "Apropiado" en este contexto significa:

- 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.

Considere las siguientes declaraciones (las funciones se marcan como `Variant 1`, `Variant 2` y `Variant 3` para su identificación en la siguiente discusión):

```
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:

```
F1 = Add( F2, 23 );
```

La instrucción anterior compila dos conjuntos:

CONJUNTO 1: FUNCIONES DE CANDIDATO CUYO PRIMER ARGUMENTO ES DE TIPO FRACCIÓN	SET 2: FUNCIONES CANDIDATAS CUYO SEGUNDO ARGUMENTO SE PUEDE CONVERTIR AL TIPO <code>INT</code>
Variante 1	Variante 1 (<code>int</code> se puede convertir en <code>long</code> mediante una conversión estándar)
Variante 3	

Las funciones del conjunto 2 son funciones para las que hay conversiones implícitas del tipo de parámetro real al tipo de parámetro formal, y entre estas funciones hay una función para la que el "costo" de convertir el tipo de parámetro real a su tipo de parámetro formal es el más pequeño.

La intersección de estos dos conjuntos es Variante 1. Un ejemplo de una llamada de función ambigua es:

```
F1 = Add( 3, 6 );
```

La llamada de función anterior compila los conjuntos siguientes:

SET 1: FUNCIONES CANDIDATAS QUE TIENEN EL PRIMER ARGUMENTO DE TIPO <code>INT</code>	CONJUNTO 2: FUNCIONES CANDIDATAS QUE TIENEN EL SEGUNDO ARGUMENTO 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 coincidencia de argumentos, una función con n argumentos predeterminados se trata como $n+1$ funciones independientes, cada una con un número diferente de argumentos.

Los puntos suspensivos (...) actúan como comodín; coinciden con cualquier argumento real. Puede conducir a muchos conjuntos ambiguos si no diseña los conjuntos de funciones sobrecargados con sumo cuidado.

NOTE

No se puede determinar la ambigüedad de las funciones sobrecargadas hasta que se encuentre 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 forma diferente al tipo base para la sobrecarga.

Sin embargo, el mecanismo de sobrecarga de funciones puede distinguir entre las referencias calificadas por `const` y `volatile` y las referencias al tipo base. Hace posible el código como el siguiente:

```

// 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

```

Over default constructor
Over&
Over default constructor
const Over&
Over default constructor
volatile Over&

```

Los punteros `const` a `volatile` objetos y también se consideran distintos de los punteros al tipo base con el fin de sobrecargarlos.

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](#)):

- Convierta de `int` a `long` y, a continuación, de `long` a `unsigned long`.
- Convertir de `int` a `unsigned long`.

La primera secuencia, aunque logra el objetivo deseado, no es la mejor secuencia coincidente: existe una secuencia más corta.

En la tabla siguiente se muestra un grupo de conversiones, denominadas conversiones triviales, que tienen un

efecto limitado en la determinación de la secuencia que se considera la mejor coincidencia. Los casos en los que las conversiones triviales influyen en la elección de la secuencia se analizan en la lista que sigue a la tabla.

Conversiones triviales

CONVERSIÓN DEL TIPO	CONVERSIÓN AL TIPO
<i>nombre de tipo</i>	<i>nombre de tipo***&</i>
<i>nombre de tipo***&*</i>	<i>nombre de tipo</i>
<i>nombre de tipo []</i>	<i>nombre de tipo*</i>
<i>type-name (lista de argumentos)</i>	<i>(* type-name) (argument-list)</i>
<i>nombre de tipo</i>	* <code>const</code> *** <i>nombre de tipo</i>
<i>nombre de tipo</i>	* <code>volatile</code> *** <i>nombre de tipo</i>
<i>nombre de tipo*</i>	* <code>const</code> *** <i>nombre de tipo*</i>
<i>nombre de tipo*</i>	* <code>volatile</code> *** <i>nombre de tipo*</i>

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. Sin embargo, las secuencias que no realizan ninguna de estas conversiones se consideran mejores que las secuencias que convierten:

- De puntero, a puntero a `const` (`type` * a `const` `type` *).
- De puntero, a puntero a `volatile` (`type` * a `volatile` `type` *).
- De referencia, para hacer referencia a `const` (`type` & a `const` `type` &).
- De referencia, para hacer referencia a `volatile` (`type` & a `volatile` `type` &).

2. Coincidencia mediante promociones. Cualquier secuencia no clasificada como coincidencia exacta que solo contenga promociones de entero, 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 `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 es tal y como se muestra en la ilustración siguiente.



Gráfico que muestra 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.

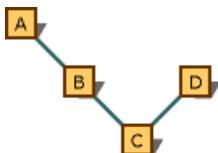


Gráfico de herencia múltiple que muestra las conversiones preferidas

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. Sin embargo, 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 rutas de acceso diferentes.

1. Coincidencia con conversiones definidas por el usuario. Esta secuencia no se puede clasificar como una coincidencia exacta, una coincidencia mediante promociones o una coincidencia mediante conversiones estándar. La secuencia debe contener solo conversiones definidas por el usuario, conversiones estándar o conversiones triviales para clasificarse como coincidencia con conversiones definidas por el usuario. 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 coteja. Tenga en cuenta el código siguiente:

```

// argument_matching1.cpp
class UDC
{
public:
    operator int()
    {
        return 0;
    }
    operator long();
};

void Print( int i )
{
};

UDC udc;

int main()
{
    Print( udc );
}

```

Las conversiones disponibles definidas por el usuario 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` y está seleccionada.

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:

```

void LogToFile( long l );
...
UDC udc;
LogToFile( udc );

```

En el ejemplo anterior, se invoca la conversión definida por el usuario, **Operator Long**, para convertir `udc` al tipo `long`. Si no se ha definido ninguna conversión definida por el usuario en `long` el tipo, la conversión se habría realizado de la siguiente manera: el tipo `UDC` se habría convertido al tipo `int` mediante la conversión definida por el usuario. A continuación, se habría aplicado la conversión estándar de tipo `int` a tipo `long` para que coincida con el argumento en la declaración.

Si es necesario que las conversiones definidas por el usuario coincidan con un argumento, las conversiones estándar no se usan al evaluar la mejor coincidencia. Incluso si más de una función candidata requiere una conversión definida por el usuario, las funciones se consideran iguales. Por ejemplo:

```

// 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 Type `int` en el argumento de tipo de clase. Las conversiones posibles son:

- Convertir tipo `int` en tipo `UDC1` (una conversión definida por el usuario).
- Convertir el tipo `int` al tipo `long`; a continuación, convertir al tipo `UDC2` (conversión en dos pasos).

Aunque el segundo requiere una conversión estándar y la conversión definida por el usuario, las dos conversiones se siguen considerando igual.

NOTE

Las conversiones definidas por el usuario se consideran conversión por construcción o conversión por inicialización (función de conversión). Ambos métodos se consideran iguales al considerar la mejor coincidencia.

Coincidencia de argumentos y el puntero this

Las funciones miembro de clase se tratan de forma diferente, dependiendo de si se declaran como `static`. Dado que las funciones no estáticas tienen un argumento implícito que proporciona el `this` puntero, se considera que las funciones no estáticas tienen un argumento más que las funciones estáticas; de lo contrario, se declaran de forma idéntica.

Estas funciones miembro no estáticas requieren que el `this` puntero implícito coincida con el tipo de objeto a través del que se llama a la función o, para los operadores sobrecargados, requieren que el primer argumento coincida con el objeto en el que se aplica el operador. (Para obtener más información sobre los operadores sobrecargados, vea [operadores sobrecargados](#)).

A diferencia de otros argumentos en funciones sobrecargadas, no se introduce ningún objeto temporal y no se intenta realizar ninguna conversión al intentar hacer coincidir el `this` argumento de puntero.

Cuando `->` se utiliza el operador de selección de miembros para tener acceso a una función miembro de clase `class_name`, el `this` argumento de puntero 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:

```
// 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 en función de si el objeto al que apunta `this` es un valor r o un valor l. Esta característica se puede usar para evitar operaciones de copia innecesarias en escenarios en los que decida no proporcionar acceso de puntero a los datos. Por ejemplo, supongamos `C` que la clase inicializa algunos datos en su constructor y devuelve una copia de los datos en la función miembro `get_data()`. Si un objeto de tipo `C` es un valor r que está a punto de ser destruido, el compilador elegirá la `get_data() &&` sobrecarga, que mueve los datos en lugar de copiarlos.

```
#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.

- Sobrecargar funciones con listas de argumentos de los mismos tipos, sobre la base exclusiva del tipo de valor devuelto, es un error.

Específicos de Microsoft

Puede sobrecargar **Operator New** únicamente en función del tipo de valor devuelto, específicamente, en función del modificador de modelo de memoria especificado.

FIN de Específicos de Microsoft

- Las funciones miembro no se pueden sobrecargar únicamente en función de una que sea estática y la otra no estática.
- **typedef** las declaraciones no definen nuevos tipos; introducen sinónimos para los tipos existentes. No afectan al mecanismo de sobrecarga. Tenga en cuenta el código siguiente:

```
typedef char * PSTR;

void Print( char *szToPrint );
void Print( PSTR szToPrint );
```

Las dos funciones anteriores tienen listas de argumentos idénticas. **PSTR** es un sinónimo del 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 con el fin de distinguir entre las funciones sobrecargadas, pero solo para las matrices de una sola dimensión. Este es el motivo por el que estas funciones sobrecargadas entran en conflicto y generan un mensaje de error:

```
void Print( char *szToPrint );
void Print( char szToPrint[] );
```

Para matrices con varias dimensiones, la segunda y todas las dimensiones sucesivas se consideran parte del tipo. Por consiguiente, se utilizan en la distinción entre funciones sobrecargadas:

```
void Print( char szToPrint[] );
void Print( char szToPrint[][7] );
void Print( char szToPrint[][9][42] );
```

Sobrecargar, reemplazar y ocultar

Dos declaraciones de función cualquiera 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; por consiguiente, 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 de la clase base, la función de clase derivada *invalida* la función de clase base. Para obtener más información, vea [funciones virtuales](#).

Si la función de clase base no está declarada como "virtual", se dice que la función de la clase derivada la *oculta*. Tanto la invalidación como la ocultación son distintas de las sobrecargas.

El ámbito de bloque es estricto. por consiguiente, una función declarada en el ámbito de archivo no está en el mismo ámbito que una función declarada localmente. 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:

```
// 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 toma un argumento de tipo `char *` es local a `main` debido a la `extern` instrucción. Por lo tanto, la definición que toma un argumento de tipo `int` está oculta y la primera llamada a `func` es 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. Se realiza mediante 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.

```

// 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;
}

```

Consulta también

[Funciones \(C++\)](#)

Funciones establecidas como valor predeterminado y eliminadas explícitamente

06/03/2021 • 12 minutes to read • [Edit Online](#)

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 lo 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.

NOTE

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 alguna razón una clase base no tiene un constructor predeterminado al que se puede llamar desde una clase derivada, es decir, un `public` `protected` constructor o que no toma ningún parámetro, una clase que deriva de él 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.

```
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 copias tiene que declararse de forma privada para ocultarlo, pero como se ha declarado en absoluto, 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.

```
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:

```
struct widget
{
    widget()=default;

    inline widget& operator=(const widget&);

    inline widget& widget::operator=(const widget&) =default;
```

Observe que puede tener como valor predeterminado una función miembro especial fuera del cuerpo de una clase siempre que sea 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 evitar que el compilador genere funciones miembro especiales que no desea. 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.

```
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.

```
// 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; }
```

Observe en el ejemplo anterior que, al llamar a mediante `call_with_true_double_only` un argumento, se `float` produciría un error del compilador, pero la llamada a con `call_with_true_double_only` un `int` argumento no lo haría; en el `int` caso de, el argumento se promoverá de `int` a `double` y llamará correctamente a la `double` versión de la función, aunque es posible que no sea lo que se pretende. Para asegurarse de que cualquier llamada a esta función con un argumento que no sea Double produce un error del compilador, puede declarar una versión de plantilla de la función que se elimina.

```
template < typename T >
void call_with_true_double_only(T) =delete; //prevent call through type promotion of any T to double from
succeding.

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

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 a una función de [postfijo](#) 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`.

```
// 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

06/03/2021 • 3 minutes to read • [Edit Online](#)

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 presentado en [sobrecarga de funciones](#).

```
// 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:

```
// 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 `print` función se cambia ligeramente para reflejar el hecho de que solo existe una de estas funciones para el tipo `double`:

```
// 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:

```
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:

```
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:

```
// 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:

```
int (*pShowIntVal)( int i = 0 );
```

Funciones insertadas (C++)

02/11/2020 • 15 minutes to read • [Edit Online](#)

Una función definida en el cuerpo de una declaración de clase es una función insertada.

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 pueden implementar como funciones insertadas.

```
// 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()
{
```

NOTE

En la declaración de clase, las funciones se han declarado sin la `inline` palabra clave. La `inline` palabra clave 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 tiene como valor predeterminado una vinculación externa a menos que una definición de esa función contenga el `inline` especificador. En el ejemplo anterior se muestra que no tiene que declarar estas funciones explícitamente con el `inline` especificador. `inline` El uso de en la definición de función hace que sea una función insertada. Sin embargo, no se permite volver a declarar una función como `inline` después de una llamada a esa función.

`inline`, `_inline` y `_forceinline`

Los `inline` `_inline` especificadores y indican al compilador que inserte una copia del cuerpo de la función en cada lugar en el que se llama a la función.

La *inserción*, llamada *expansión en línea o inline*, solo se produce si el análisis de costos y beneficios del compilador muestra que merece la pena. La expansión en línea minimiza la sobrecarga de la llamada de función en el costo potencial de mayor tamaño del código.

La `_forceinline` palabra clave invalida el análisis de costos y beneficios y se basa en la sentencia del programador en su lugar. Tenga cuidado al usar `_forceinline`. El uso indiscriminado de `_forceinline` puede dar lugar a código más grande, con solo mejoras marginales en el rendimiento o, en algunos casos, incluso pérdidas de rendimiento (debido al aumento de la paginación de un archivo ejecutable más grande, por ejemplo).

El uso de funciones insertadas puede agilizar la ejecución del programa porque eliminan la sobrecarga asociada a las llamadas a función. Las funciones expandidas insertadas están sujetas a optimizaciones de código que no están disponibles para las funciones normales.

El compilador trata las opciones de expansión insertada y las palabras clave como sugerencias. No hay ninguna garantía de que las funciones se insertarán. No se puede forzar al compilador para que inlinee una función determinada, incluso con la `_forceinline` palabra clave. Al compilar con `/c1r`, el compilador no insertará una función si hay atributos de seguridad aplicados a la función.

La `inline` palabra clave solo está disponible en C++. Las `_inline` `_forceinline` palabras clave y están disponibles tanto en C como en C++. Por compatibilidad con versiones anteriores, `_inline` y `_forceinline` son sinónimos para `_inline`, y `_forceinline` a menos que se especifique la opción del compilador [/Za](#) ([deshabilitar extensiones de lenguaje](#)).

La `inline` palabra clave indica al compilador que se prefiere la expansión en línea. Sin embargo, el compilador puede crear una instancia independiente de la función y crear vinculaciones de llamada estándar en lugar de insertar el código. Dos casos en los que puede producirse este comportamiento son:

- Funciones recursivas.
- Funciones a las que se hace referencia mediante un puntero en otra parte de la unidad de traducción.

Estas razones pueden interferir con la inserción, *como pueden ser otras*, a discreción del compilador. no debe depender del `inline` especificador para hacer que una función esté insertada.

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 pasa mediante el protocolo normal de llamada de función.

La `/Ob` opción de optimización del compilador ayuda a determinar si realmente se produce la expansión de la función insertada.

[/LTCG](#) realiza la inserción entre módulos, tanto si se solicita en código fuente como si no lo está.

Ejemplo 1

```
// 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 en línea, ya sea mediante la `inline` palabra clave o colocando la definición de función dentro de la definición de clase.

Ejemplo 2

```
// 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 `__inline` palabra clave es equivalente a `inline`.

Incluso con `__forceinline`, el compilador no puede alinear código en todas las circunstancias. El compilador no puede alinear 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 alineado, a menos que se compile con `/Ox`, `/O1` o `/O2`.
- La función es recursiva y no tiene `#pragma inline_recursion(on)` establecido. Con la directiva pragma, las funciones recursivas se alinean hasta una profundidad predeterminada de 16 llamadas. Para reducir la profundidad de inserción, utilice `inline_depth` pragma.
- 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 también se marca con el `naked` `__declspec` modificador.

Si el compilador no puede alinear 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 indirecta. El compilador no puede identificar código no insertado que no se encuentre en la unidad de traducción actual.

Las funciones recursivas se pueden reemplazar por código alineado a una profundidad especificada por la `inline_depth` Directiva pragma, 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 se examinan las funciones recursivas mediante la heurística alineada no puede ser superior a 16. La `inline_recursion` Directiva pragma controla la expansión alineada de una función que se encuentra actualmente en expansión.

Consulte la opción del compilador de [expansión de funciones insertadas](#) (/OB) para obtener información relacionada.

FIN de Específicos de Microsoft

Para obtener más información sobre el uso del `inline` especificador, 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 las llamadas a funciones. Las funciones más largas pasan proporcionalmente menos tiempo en la secuencia de llamada y devolución, y se benefician menos de la inserción.

Una `Point` clase se puede definir de la siguiente manera:

```
// 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()
{}
```

Suponiendo que la manipulación de coordenadas es una operación relativamente común en un cliente de dicha clase, especificando las dos funciones de descriptor de acceso (`x` y `y` en el ejemplo anterior), ya que `inline` normalmente guarda 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 de marco de pila
- Comunicación de valor devuelto
- Restaurar el 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 las macros las expande el preprocesador. 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 `inline` palabra clave 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:

```
// 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
```

La intención de la expresión `toupper(getc(stdin))` es que se debe leer un carácter desde el dispositivo de consola (`stdin`) y, si es necesario, convertirlo 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 intervalo, `getc` se ejecuta de nuevo para convertir el carácter a mayúsculas. Significa que el programa espera dos o tres caracteres cuando, idealmente, debería esperar solo uno.

Las funciones insertadas solucionan el problema descrito anteriormente:

```
// 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 );
}
```

```
Sample Input: a
Sample Output: A
```

Consulte también

[noinline](#)

[auto_inline](#)

Sobrecarga de operadores

06/03/2021 • 5 minutes to read • [Edit Online](#)

La `operator` palabra clave declara una función que especifica qué significa el *símbolo del 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 de Operator-Symbol(lista de parámetros)`

Observaciones

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 y 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, `+ =`, defina una función llamada `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
()	Llamada a función	—
()	Operador de conversión	Unario
*	Multiplicación	Binary

OPERATOR	NOMBRE	TIPO
*	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
>>=	Asignación y desplazamiento a la derecha	Binary

OPERATOR	NOMBRE	TIPO
[]	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 unarios de incremento y decremento: preincremento y postincremento.

Vea [reglas generales para 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](#)
- [Asignación](#)
- [Llamada de 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

OPERATOR	NOMBRE
? :	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:

```
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.

```
// 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();
}
```

6.8, 11.2

En esta sección

- [Reglas generales para la sobrecarga de operadores](#)
- [Sobrecargar operadores unarios](#)
- [Operadores binarios](#)
- [Asignación](#)

- [Llamada de función](#)
- [Subíndices](#)
- [Acceso a miembros](#)

Consulta también

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

[Palabras clave](#)

Reglas generales para la sobrecarga de operadores

06/03/2021 • 5 minutes to read • [Edit Online](#)

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 describen por separado.

- No se pueden definir nuevos operadores, como ..
- 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:

```
// 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 commutativa. Es tan probable que se implementen los 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 lo tanto, no hay ninguna manera de expresar el concepto "agregar 2 y 3 a un objeto de tipo `Point`", se espera que 2 se agregue a la coordenada `x` y 3 que se agregue 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 se puede usar un operador como operador unario o binario (`&`, `*`, `+` y `-`), puede sobrecargar cada uso por separado.
- Los operadores sobrecargados no pueden tener argumentos predeterminados.
- Todas las clases derivadas heredan todos los operadores sobrecargados excepto la asignación (operador

=).

- 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 Call-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:

```
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/asignación, += , requiere que el operando izquierdo sea un valor l cuando se aplica a los tipos básicos; no existe este requisito cuando el operador está sobrecargado.

NOTE

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.

Consulta también

[Sobrecarga de operadores](#)

Sobrecargar operadores unarios

06/03/2021 • 2 minutes to read • [Edit Online](#)

Los operadores unarios que se pueden sobrecargar son los siguientes:

1. `!` (Not lógico)
2. `&` (dirección de)
3. `~` (complemento de uno)
4. `*` (desreferencia de puntero)
5. `+` (unario más)
6. `-` (negación unaria)
7. `++` (incremento)
8. `--` (decremento)
9. operadores de conversión

Los operadores de incremento y decremento de postfijo (`++` y `--`) se tratan por separado en [incremento y decremento](#).

Los operadores de conversión también se tratan en un tema independiente; vea [conversiones de tipos definidos por el usuario](#).

Las reglas siguientes son ciertas para todos los demás operadores unarios. Para declarar una función de operador unario como miembro no estático, debe declararla de la forma siguiente:

```
RET-tipo operator OP()
```

donde *RET-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 función global, debe declararla de la forma siguiente:

```
RET-tipo operator OP( arg )
```

donde *RET-Type* y *OP* son como se describen para las funciones de operador de miembro y *arg* es un argumento de tipo de clase en el que se va a operar.

NOTE

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 entero, pero esto no siempre sucede así.

Consulta también

[Sobrecarga de operadores](#)

Sobrecarga de operadores de incremento y decremento (C++)

06/03/2021 • 3 minutes to read • [Edit Online](#)

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 escribe funciones de operador sobrecargadas, puede resultarle útil implementar versiones distintas para las versiones de prefijo y de postfijo de estos operadores. Para distinguir entre los dos, se observa la siguiente regla: el formato de prefijo del operador se declara exactamente de la misma manera que cualquier otro operador unario; el formato de postfijo acepta un argumento adicional de tipo `int`.

NOTE

Al especificar un operador sobrecargado para la forma de postfijo del operador de incremento o decrecimiento, el argumento adicional debe ser de tipo; si se `int` especifica cualquier otro tipo, se genera un error.

En el ejemplo siguiente se muestra cómo definir operadores de incremento y decrecimiento de prefijo y de postfijo para la clase `Point`:

```

// 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 encabezados de función:

```

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 denota la forma de postfijo del operador de incremento o decremento no se usa normalmente para pasar argumentos. Normalmente contiene el valor 0. Sin embargo, se puede utilizar del modo siguiente:

```
// increment_and_decrement2.cpp
class Int
{
public:
    Int &operator++( int n );
private:
    int _i;
};

Int& Int::operator++( int n )
{
    if( n != 0 )    // Handle case where an argument is passed.
        _i += n;
    else
        _i++;       // Handle case where no argument is passed.
    return *this;
}
int main()
{
    Int i;
    i.operator++( 25 ); // Increment by 25.
}
```

No hay ninguna sintaxis 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 y asignación (`+=`).

Consulta también

[Sobrecarga de operadores](#)

Operadores binarios

06/03/2021 • 2 minutes to read • [Edit Online](#)

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
<=	Menor o igual que
=	Asignación
==	Igualdad
>	Mayor que
>=	Mayor o igual que
>>	Desplazamiento a la derecha
>>=	Desplazamiento a la derecha/asignación
^	OR exclusivo
^=	OR exclusivo/asignación
	OR inclusivo bit a bit
=	OR inclusivo bit a bit/asignación
	O lógico

Para declarar una función de operador binario como miembro no estático, debe declararla de la forma siguiente:

```
RET-tipo operator OP( arg)
```

donde *RET-Type* es el tipo de valor devuelto, *OP* es uno de los operadores enumerados 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-tipo operator OP( arg1, arg2)
```

donde *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.

NOTE

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.

Consulta también

[Sobrecarga de operadores](#)

Asignación

06/03/2021 • 2 minutes to read • [Edit Online](#)

El operador de asignación (=) es, en realidad, un operador binario. La declaración es idéntica a cualquier otro operador binario, con las excepciones siguientes:

- Debe ser una función miembro no estática. No se puede declarar ningún **operador =** como una 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:

```
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:

```
pt1 = pt2 = pt3;
```

El operador de asignación de copia no se debe confundir con el constructor de copias. Se llama a este último durante la construcción de un nuevo objeto a partir de uno existente:

```
// 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.
```

NOTE

Es aconsejable seguir la [regla de tres](#) que una clase que define un operador de asignación de copia también debe definir explícitamente el constructor de copias, el destructor y, a partir de C++ 11, el constructor de movimiento y el operador de asignación de movimiento.

Consulta también

- [Sobrecarga de operadores](#)
- [Constructores de copia y operadores de asignación de copia \(C++\)](#)

Llamada de función (C++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

El operador de llamada a función, invocado mediante paréntesis, es un operador binario.

Sintaxis

```
primary-expression ( expression-list )
```

Observaciones

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:

```
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:

```
// 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).

```
typedef void(*ptf)();
void func()
{
}
struct S
{
    operator ptf()
    {
        return func;
    }
};

int main()
{
    S s;
    s(); //operates as s.operator ptf()()
}
```

Consulta también

[Sobrecarga de operadores](#)

Subíndices

06/03/2021 • 2 minutes to read • [Edit Online](#)

El operador de subíndice ([]), al igual que el operador de llamada a 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:

```
// 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;
}
```

```
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, el **operador []** detecta que se está usando un subíndice fuera del límite y emite un mensaje de error.

Tenga en cuenta que el operador de función `[]` 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.

Consulta también

[Sobrecarga de operadores](#)

Acceso a miembros

06/03/2021 • 2 minutes to read • [Edit Online](#)

El acceso a miembros de clase se puede controlar sobrecargando el 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->()
```

Observaciones

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 elemento de lenguaje `.` no se puede sobrecargar el operador de acceso a miembros.

Consulta también

[Sobrecarga de operadores](#)

Clases y structs (C++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

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](#)
- [Access Control miembro](#)
- [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](#)
- [Puntero this](#)
- [Campos de bits de C++](#)

Los tres tipos de clase son estructura, clase, y unión. Se declaran mediante las palabras clave [struct](#), [Classy Union](#). En la tabla siguiente se muestran las diferencias entre los tres tipos de clase.

Para obtener más información sobre las uniones, vea [uniones](#). Para obtener información sobre las clases y los Structs en C++/CLI y C++/CX, vea [clases y Structs](#).

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

class (C++)

06/03/2021 • 3 minutes to read • [Edit Online](#)

La `class` palabra clave 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

Plantilla: Especificación

Especificaciones de plantilla opcionales. Para obtener más información, consulte [plantillas](#).

class

`class` Palabra clave.

MS-decl-Spec

Especificación opcional de clase de almacenamiento. Para obtener más información, consulte la palabra clave [__declspec](#).

etiqueta

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, vea [tipos de clase anónimos](#).

base-list

Lista opcional de clases o de estructuras de las que esta clase derivará sus miembros. Para obtener más información, vea [clases base](#). Cada nombre de clase base o estructura puede ir precedido de un especificador de acceso ([Public](#), [Private](#), [Protected](#)) y la palabra clave [virtual](#). Vea la tabla de acceso a miembros en [controlar el acceso a miembros de clase](#) para obtener más información.

lista de miembros

Lista de miembros de clase. Consulte [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 común en estructuras, cuyos miembros de datos son de `public` forma predeterminada, que en las clases. Vea [información general sobre declaradores](#) para obtener más información.

Observaciones

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 los Structs administrados en C++/CLI y C++/CX, vea [clases y Structs](#)

Ejemplo

```
// 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++)

06/03/2021 • 4 minutes to read • [Edit Online](#)

La `struct` palabra clave 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

Plantilla: Especificación

Especificaciones de plantilla opcionales. Para obtener más información, consulte [Especificaciones de plantilla](#).

struct

`struct` Palabra clave.

MS-decl-Spec

Especificación opcional de clase de almacenamiento. Para obtener más información, consulte la palabra clave [_declspec](#).

etiqueta

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 obtener más información, vea [tipos de clase anónimos](#).

base-list

La lista opcional de clases o de estructuras de la que esta estructura derivará sus miembros. Para obtener más información, vea [clases base](#). Cada nombre de clase base o estructura puede ir precedido de un especificador de acceso ([Public](#), [Private](#), [Protected](#)) y la palabra clave [virtual](#). Vea la tabla de acceso a miembros en [controlar el acceso a miembros de clase](#) para obtener más información.

lista de miembros

Lista de miembros de la estructura. Consulte [información general sobre miembros de clase](#) para obtener más información. La única diferencia aquí es que `struct` se utiliza en lugar de `class`.

declarators

Lista de declaradores que especifica 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 la estructura son `public`. Las listas de inicializadores son comunes en estructuras porque los miembros de datos son `public` de forma predeterminada. Vea [información general sobre declaradores](#) para obtener más información.

Observaciones

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 los Structs administrados en C++/CLI, vea [clases y Structs](#).

Uso de una estructura

En C, debe utilizar explícitamente la `struct` palabra clave para declarar una estructura. En C++, no es necesario usar la `struct` palabra clave 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, vea [clase, Unióny enumeración](#).

Ejemplo

```
#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; // ?000000 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

06/03/2021 • 6 minutes to read • [Edit Online](#)

Una clase o struct está compuesta por 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 realiza mediante constructores y el trabajo de limpieza, como la liberación de memoria y la liberación de 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 las funciones miembro](#).
- [Miembros de datos](#), incluidos los tipos integrados y otros tipos definidos por el usuario.
- Operadores
- [Declaraciones de clase anidada](#) y.)
- [Uniones](#)
- [Enumeraciones](#).
- [Campos de bits](#).
- [Amigos](#).
- [Alias y typedefs](#).

NOTE

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:

```

// 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 cualquier número `private` de `protected` secciones, y `public` mediante el uso de palabras clave conocidas como especificadores de acceso. Dos puntos : deben seguir el especificador de acceso. Estas secciones no necesitan 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 obtener más información, vea [Access Control de miembro \(C++\)](#).

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 se puede declarar como estática, en cuyo caso solo puede tener acceso a los miembros de datos estáticos de la clase (y no tiene *este* puntero). Para obtener más información, vea [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.

1. Constructor predeterminado
2. Constructor de copias
3. (C++ 11) Constructor de movimiento
4. Operador de asignación de copia
5. (C++ 11) Operador de asignación de movimiento
6. Destructor

Para obtener más información, vea [funciones miembro especiales](#).

Inicialización miembro a miembro

En C++11 y versiones posteriores, los declaradores de miembros no estáticos pueden contener inicializadores.

```
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 con el que se inicializó el miembro en el punto de 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 obtener más información acerca de los miembros de datos estáticos, vea [miembros de datos estáticos](#)). En el ejemplo siguiente se muestra cómo realizar estas inicializaciones:

```
// 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;
```

NOTE

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++)

06/03/2021 • 11 minutes to read • [Edit Online](#)

Los controles de acceso permiten separar la interfaz **pública** de una clase de los detalles de implementación **privados** y los miembros **protegidos** que solo se usan en 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.

```
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 está **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 objetos de tipos de clase depende de la implementación, pero se garantiza que a los miembros se les asignen direcciones de memoria sucesivamente superiores entre los especificadores de acceso.

Control de acceso a miembros

TIPO DE ACCESO	SIGNIFICADO
private	Los miembros de clase declarados como private solo se pueden usar en funciones miembro y amigos (clases o funciones) de la clase.
protected	Los miembros de clase declarados como protected pueden ser utilizados por funciones miembro y amigos (clases o funciones) de la clase. Además, las clases derivadas de la clase también pueden usarlos.
public	Cualquier función puede usar los miembros de clase declarados como public .

El control de acceso impiden usar objetos de manera diferente a la que están destinados. Esta protección se pierde cuando se realizan conversiones de tipos explícitas.

NOTE

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 mediante 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

PRIVATE	PROTECTED	PÚBLICO
Siempre inaccesible independientemente del acceso de derivación	Privado en la clase derivada si se utiliza la derivación privada	Privado en la clase derivada si se utiliza la derivación privada
	Protegido en la clase derivada si se utiliza la derivación protegida	Protegido en la clase derivada si se utiliza la derivación protegida
	Protegido en la clase derivada si se utiliza la derivación pública	Público en la clase derivada si se utiliza la derivación pública

Esto se ilustra en el ejemplo siguiente:

```

// 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 miembro público y `ProtectedFunc` es un miembro protegido porque `BaseClass` es una clase base pública. `PrivateFunc` es privado para `BaseClass`, y es inaccesible en cualquiera de sus clases derivadas.

En `DerivedClass2`, las funciones `PublicFunc` y `ProtectedFunc` se consideran miembros privados porque `BaseClass` es una clase base privada. De nuevo, `PrivateFunc` es privado para `BaseClass`, y es inaccesible en cualquiera de sus clases derivadas.

Puede declarar una clase derivada sin un especificador de acceso de clase base. En tal caso, la derivación se considera privada si la declaración de clase derivada utiliza la `class` palabra clave. La derivación se considera pública si la declaración de clase derivada utiliza la `struct` palabra clave. Por ejemplo, el código siguiente:

```

class Derived : Base
...

```

equivale a:

```

class Derived : private Base
...

```

De igual forma, el código siguiente:

```
struct Derived : Base  
{
```

equivale a:

```
struct Derived : public Base  
{
```

Tenga en cuenta que los miembros declarados como con acceso privado no son accesibles para funciones o clases derivadas, a menos que esas funciones o clases se declaren mediante la `friend` declaración en la clase base.

Un `union` tipo no puede tener una clase base.

NOTE

Al especificar una clase base privada, es aconsejable usar explícitamente la `private` palabra clave para que los usuarios de la clase derivada sepan el acceso a 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 el uso de punteros, referencias u objetos puede requerir una conversión en la que se aplica de nuevo el control de acceso de tiempo. Considere el ejemplo siguiente:

```
// 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. Conversion of  
                            // Derived2 * to Base * not  
                            // permitted.  
    return cCount;  
}
```

En el código anterior, el control de acceso prohíbe la conversión de un puntero a `Derived2` en un puntero a

`Base`. El `this` puntero es implícitamente de tipo `Derived2 *`. Para seleccionar la `Countof` función, `this` se debe convertir al tipo `Base *`. Este tipo de conversión no está permitido porque `Base` es una clase base indirecta privada a `Derived2`. La conversión a un tipo de clase base privada solo es aceptable para punteros a clases derivadas inmediatas. Por lo tanto, los punteros de tipo `Derived1 *` pueden convertirse al tipo `Base *`.

Observe que la llamada explícita a la función `Countof`, sin utilizar un puntero, una referencia o un objeto para seleccionarla, no implica ninguna conversión. Por lo tanto, se permite la llamada.

Los miembros y objetos friend de una clase derivada, `T`, pueden convertir un puntero a `T` en un puntero a una clase base directa privada de `T`.

Acceso a funciones virtuales

El control de acceso que se aplica a las funciones [virtuales](#) viene determinado por el tipo que se usa para hacer la llamada de función. El reemplazo de las declaraciones de la función no afecta al control de acceso para un tipo determinado. Por ejemplo:

```
// 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 *pvd = &vfd;  // Pointer to derived type.
    int State;

    State = pvfb->GetState(); // GetState is public.
    State = pvd->GetState();  // C2248 error expected; GetState is private;
}
```

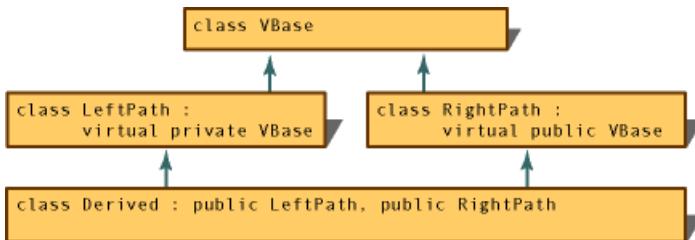
En el ejemplo anterior, al llamar a la función virtual `GetState` mediante un puntero al tipo `VFuncBase`, se llama a `VFuncDerived::GetState`, y `GetState` se trata como public. Sin embargo, la llamada a `GetState` mediante un puntero al tipo `VFuncDerived` se considera una infracción del control de acceso, porque `GetState` se declara `private` en la clase `VFuncDerived`.

Caution

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`. La ruta de acceso derecha es más accesible porque `RightPath` declara `VBase` como clase base pública, mientras que `LeftPath` declara `VBase` como private.

Vea también

[Referencia del lenguaje C++](#)

friend (C++)

06/03/2021 • 10 minutes to read • [Edit Online](#)

En algunas circunstancias, es más conveniente conceder acceso de nivel de miembro a las funciones que no son miembros de una clase o a todos los miembros de una clase independiente. 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 `friend` palabra clave y el nombre de una función no miembro u otra clase para concederle acceso a los miembros privados y protegidos de la clase. En una definición de plantilla, un parámetro de tipo puede declararse como Friend.

Sintaxis

```
class friend F  
friend F;
```

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 `extern` palabra clave. Para obtener 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 siguiente ejemplo de código se muestra por qué esto produce un error:

```
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 lo tanto, la `friend` declaración en la clase `HasFriends` genera un error.

A partir de C++ 11, hay dos formas de declaraciones Friend para una clase:

```
friend class F;  
friend F;
```

El primer formulario introduce una nueva clase F si no se encontró ninguna clase existente con ese nombre en el espacio de nombres más interno. C++ 11: el segundo formulario no introduce una nueva clase; se puede usar cuando la clase ya se ha declarado y debe usarse al declarar un parámetro de tipo de plantilla o una definición de tipo como Friend.

Use `class friend F` cuando el tipo al que se hace referencia aún no se ha declarado:

```
namespace NS
{
    class M
    {
        class friend F; // Introduces F but doesn't define it
    };
}
```

```
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 `F` clase declarada fuera del ámbito de NS.

```
class F {};
namespace NS
{
    class M
    {
        friend F; // OK
    };
}
```

Use `friend F` para declarar un parámetro de plantilla como Friend:

```
template <typename T>
class my_class
{
    friend T;
    //...
};
```

Utilice `friend F` para declarar una definición de tipo como Friend:

```
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.

NOTE

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 `friend` función 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. Los amigos no se encuentran en el ámbito de la clase y no se llaman mediante los operadores de selección de miembro (`. y ->`) a menos que sean miembros de otra clase. Una `friend` función se declara mediante la clase que concede el acceso. La `friend` declaración se puede colocar en cualquier parte 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 `friend` función tiene acceso al miembro de datos privado del `Point` objeto que recibe como parámetro.

```
// 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:

```

// 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 de confianza a la clase `B`. Por lo tanto, el acceso al miembro privado `_b` es correcto en la `Func1` clase, `A` pero no en `Func2`.

Una `friend` clase es una clase de cuyas funciones miembro son 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 `friend` declaración de la clase `B` ha sido:

```
friend class A;
```

En ese caso, a todas las funciones miembro de la clase `A` se les habría concedido acceso de confianza a la clase `B`. El código siguiente es un ejemplo de una clase de confianza:

```

// 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();
}

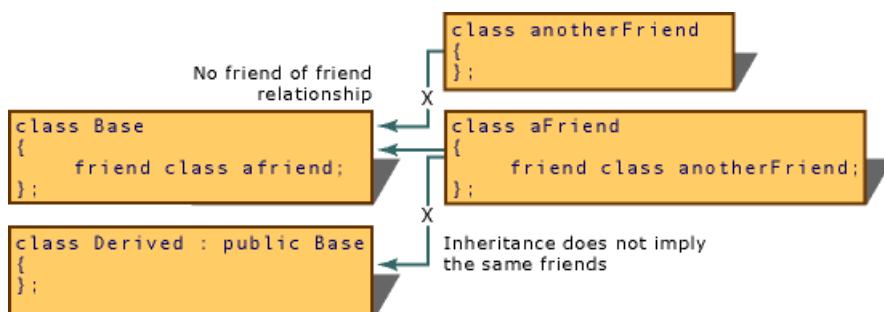
```

La confianza 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 ninguna función Friend, clases Friend ni interfaces Friend.

La confianza no se hereda, lo que significa que las clases derivadas de `YourOtherClass` no pueden tener acceso a los miembros privados de `YourClass`. La confianza no es transitiva, por lo que las clases de confianza 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

Definiciones friend en línea

Las funciones Friend se pueden definir (dado un cuerpo de función) dentro de las declaraciones de clase. Estas funciones son funciones insertadas y 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 las declaraciones de clase están en el ámbito de la clase envolvente.

Consulta también

[Palabras clave](#)

private (C++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

Sintaxis

```
private:  
    [member-list]  
private base-class
```

Observaciones

Cuando precede a una lista de miembros de clase, la `private` palabra clave especifica que esos miembros son accesibles solo desde las funciones miembro y amigos de la clase. Esto se aplica a todos los miembros declarados hasta el especificador de acceso siguiente o el final de la clase.

Al anteponer el nombre de una clase base, la `private` palabra clave 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, vea [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 CLR, las palabras clave del especificador de acceso de C++ (`public`, `private` y `protected`) pueden afectar a la visibilidad de los tipos y métodos con respecto a los ensamblados. Para obtener más información, vea [Access Control miembro](#).

NOTE

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

```
// 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++)

06/03/2021 • 3 minutes to read • [Edit Online](#)

Sintaxis

```
protected:  
    [member-list]  
protected base-class
```

Observaciones

La `protected` palabra clave especifica el acceso a miembros de clase en la *lista de miembros* hasta el siguiente especificador de acceso (`public` o `private`) o el final de la definición de clase. Los miembros de clase declarados como `protected` solo se pueden usar por lo siguiente:

- 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.

Al anteponer el nombre de una clase base, la `protected` palabra clave 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 `private` los miembros, a los que solo se puede tener acceso a los miembros de la clase en la que se declaran, pero no son tan públicos como `public` miembros, que son accesibles en cualquier función.

Los miembros protegidos que también se declaran como `static` son accesibles para cualquier función Friend o miembro de una clase derivada. Los miembros protegidos que no se declaran como `static` son accesibles para los amigos y las funciones miembro de una clase derivada solo a través de un puntero a, referencia a u objeto de la clase derivada.

Para obtener información relacionada, vea [Friend](#), [Public](#), [Private](#) y la tabla de acceso a miembros en [controlar el acceso a los miembros de clase](#).

Específicos de /clr

En los tipos CLR, las palabras clave del especificador de acceso de C++ (`public`, `private` y `protected`) pueden afectar a la visibilidad de los tipos y métodos con respecto a los ensamblados. Para obtener más información, vea [Access Control miembro](#).

NOTE

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

```
// 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++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

Sintaxis

```
public:  
  [member-list]  
public base-class
```

Observaciones

Cuando precede a una lista de miembros de clase, la `public` palabra clave 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 `public` palabra clave 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, vea [Private, Protected, Friendly](#) la tabla de acceso a miembros en [controlar el acceso a los miembros de clase](#).

Específicos de /clr

En los tipos CLR, las palabras clave del especificador de acceso de C++ (`public`, `private` y `protected`) pueden afectar a la visibilidad de los tipos y métodos con respecto a los ensamblados. Para obtener más información, vea [Access Control miembro](#).

NOTE

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

```
// 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

02/11/2020 • 5 minutes to read • [Edit Online](#)

No siempre es necesario definir un constructor para una clase, especialmente si son relativamente sencillas. Los usuarios pueden inicializar objetos de una clase o struct usando la inicialización uniforme, tal y como se muestra en el siguiente ejemplo:

```
// 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 };

    // Default initialization = {0,0,0,0,0}
    TempData td_default{};

    // 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;
}
```

Tenga en cuenta que cuando una clase o estructura no tiene ningún constructor, se proporcionan los elementos de la lista en el orden en que se declaran los miembros en la clase. Si la clase 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 de forma explícita, puede utilizar la inicialización de llave predeterminada (con las llaves vacías). Por ejemplo, la siguiente clase puede inicializarse mediante la inicialización de llave predeterminada y la no predeterminada:

```

#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, el orden en que los miembros aparecen en el inicializador de llave es el mismo que el orden en que se declararon; 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:

```

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 predeterminada:

```

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 usar la inicialización de llaves en cualquier lugar en el que normalmente realizaría la inicialización; por ejemplo, como un parámetro de función o un valor devuelto, o con la `new` palabra clave:

```

class_d* cf = new class_d{4.5};
kr->add_d({ 4.5 });
return { 4.5 };

```

En el modo `/STD: c++ 17`, las reglas para la inicialización de llave vacía son ligeramente más restrictivas. Vea [constructores derivados e inicialización de agregados extendidos](#).

constructores initializer_list

La [clase initializer_list](#) representa una lista de objetos de un tipo especificado que se pueden usar en un constructor y en otros contextos. Puede construir `initializer_list` mediante la inicialización de llave:

```
initializer_list<int> int_list{5, 6, 7};
```

IMPORTANT

Para usar esta clase, debe incluir el `<initializer_list>` encabezado.

`initializer_list` puede copiarse. En este caso, los miembros de la nueva lista son referencias a los miembros de la lista original:

```

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:

```

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)

21/03/2020 • 5 minutes to read • [Edit Online](#)

A diferencia de los lenguajes administrados, no tiene recolección automática de C++ *elementos no utilizados*. Este es un proceso interno que libera la memoria del montón y otros recursos a medida que se ejecuta un programa. Un C++ programa es responsable de devolver todos los recursos adquiridos al sistema operativo. Un error al liberar un recurso sin usar se denomina *fuga*. Los recursos filtrados no están disponibles para otros programas hasta que el proceso se cierra. En concreto, las pérdidas de memoria son una causa común de errores en la programación de estilo C.

Moderno C++ evita usar la memoria del montón lo máximo posible declarando 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. Después, 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 propios recursos* también se conoce como "adquisición de recursos," o RAII.

Cuando un objeto de pila propietario del recurso sale del ámbito, se invoca automáticamente su destructor. De esta manera, la recolección de C++ elementos no utilizados en está estrechamente relacionada con la duración de los objetos y es determinista. Un recurso siempre se libera en un punto conocido del programa, que se puede controlar. Solo los destructores deterministas, como C++ los de, pueden controlar de forma equitativa los recursos de memoria y no de memoria.

En el ejemplo siguiente 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 `g` miembro se declara en la pila y simplemente sale del ámbito junto con `w`. No se necesita ningún código especial en el destructor `widget`.

```
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.

```

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 manera mejor de escribir el ejemplo anterior: mediante el uso de un puntero inteligente de la biblioteca estándar. El puntero inteligente controla la asignación y eliminación de la memoria que posee. El uso de un puntero inteligente elimina la necesidad de un destructor explícito en la clase `widget`.

```

#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 posible pérdida de memoria. Este modelo funciona para otros recursos, como identificadores de archivo o Sockets. Puede administrar sus propios recursos de una manera similar en sus clases. Para obtener más información, vea [punteros inteligentes](#).

El diseño de C++ garantiza que los objetos se destruyen cuando salen del ámbito. Es decir, se destruyen cuando se cierran los bloques, en orden inverso a la 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 de depurar si el constructor de un objeto global produce una excepción.

Consulte también

[Bienvenido de nuevo a C++](#)

[Referencia del lenguaje C++](#)

[Biblioteca estándar de C++](#)

Pimpl para encapsulación en tiempo de compilación (C++ moderno)

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 implementación". Es posible que ya esté familiarizado con el concepto, pero lo sepa con otros nombres como Cheshire cat o la expresión de firewall del compilador.

¿Por qué usar pimpl?

A continuación se muestra cómo la expresión *pimpl* puede mejorar el ciclo de vida de desarrollo de software:

- Minimización de las dependencias de compilación.
- Separación de la interfaz y la implementación.
- Portabilidad.

Encabezado Pimpl

```
// 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 la recompilación en cascada y los diseños de objeto frágiles. Es adecuado para los tipos populares (transitivamente).

Implementación de Pimpl

Defina la `impl` clase en el archivo .cpp.

```
// 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 la posibilidad de agregar compatibilidad para la especialización de intercambio que no inicia.

Consulta también

[Aquí está otra vez C++](#)

Portabilidad en los límites de ABI

02/12/2019 • 2 minutes to read • [Edit Online](#)

Utilice tipos y convenciones suficientemente portátiles en los límites de la interfaz binaria. Un "tipo portátil" es un tipo integrado de C o un struct que solo contiene tipos integrados de C. Los tipos de clase solo se pueden usar cuando el llamador y el destinatario acuerdan el diseño, la Convención de llamada, etc. Esto solo es posible cuando ambos se compilan con la misma configuración del compilador y del compilador.

Cómo aplanar una clase para la portabilidad de C

Cuando los autores de la llamada se pueden compilar con otro lenguaje o compilador, "aplanar" en una API de **"C" externa** con una Convención de llamada específica:

```
// 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"  
}
```

Vea también

[Bienvenido de nuevo a C++](#)

[Referencia del lenguaje C++](#)

[Biblioteca estándar de C++](#)

Constructores (C++)

06/03/2021 • 30 minutes to read • [Edit Online](#)

Para personalizar cómo se inicializan los miembros de clase 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 sean necesarios para personalizar la inicialización de varias maneras. Normalmente, los constructores tienen accesibilidad pública para que el código fuera de la definición de clase o de la jerarquía de herencia pueda crear objetos de la clase. Pero también puede declarar un constructor como `protected` o `private`.

Opcionalmente, los constructores pueden tomar una lista de miembros init. Se trata de una manera más eficaz de inicializar miembros de clase que asignar valores en el cuerpo del constructor. En el ejemplo siguiente se muestra una clase `Box` con tres constructores sobrecargados. Las dos últimas listas de inicialización de miembros de uso:

```
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 };
};
```

Cuando se declara una instancia de una clase, el compilador elige el constructor que se va a invocar en función de las reglas de resolución de sobrecarga:

```
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 declarado como `const` `volatile` o `const volatile`. El objeto se convierte en `const` una vez que se completa el constructor.
- Para definir un constructor en un archivo de implementación, asígnele un nombre completo como con cualquier otra función miembro: `Box::Box(){...}`.

Listas de inicializadores de miembro

Un constructor puede tener opcionalmente una lista de inicializadores de miembro, que inicializa los miembros de clase antes de la ejecución del cuerpo del constructor. (Tenga en cuenta que una lista de inicializadores de miembro no es lo mismo que una *lista de inicializadores* de tipo `std <T> :: initializer_list`).

El uso de una lista de inicializadores de miembro es preferible a la asignación de valores en el cuerpo del constructor porque inicializa directamente el miembro. En el ejemplo siguiente se muestra que la lista de inicializadores de miembro consta de todas las expresiones de **identificador (argumento)** detrás del signo de dos puntos:

```
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 clase. se inicializa con el valor del argumento. El argumento puede ser uno de los parámetros de constructor, una llamada de función o un `STD: <T> : initializer_list`.

`const` los miembros y miembros de tipo de referencia se deben inicializar en la lista de inicializadores de miembro.

Las llamadas a los constructores de clase base con parámetros deben realizarse en la lista de inicializadores para asegurarse de que la clase base se inicializa completamente antes de la ejecución del constructor derivado.

Constructores predeterminados

Normalmente, los *constructores predeterminados* no tienen parámetros, pero pueden tener parámetros con valores predeterminados.

```
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 `inline` constructor predeterminado implícito.

```

#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 confía en un constructor predeterminado implícito, asegúrese de inicializar los miembros en la definición de clase, tal como se muestra en el ejemplo anterior. Sin esos inicializadores, los miembros se desinicializarían y la llamada de volumen () generaría un valor de elementos no utilizados. En general, se recomienda inicializar los miembros de esta manera incluso cuando no se confía en un constructor predeterminado implícito.

Puede evitar que el compilador genere un constructor predeterminado implícito si lo define como [eliminado](#):

```

// Default constructor
Box() = delete;

```

Un constructor predeterminado generado por el compilador se definirá como eliminado si alguno de los miembros de clase no es constructor predeterminado. Por ejemplo, todos los miembros del tipo de clase y sus miembros de tipo de clase deben tener un constructor y destructores predeterminados a los que se pueda tener acceso. Todos los miembros de datos del tipo de referencia, así como [const](#) los miembros deben tener un inicializador de miembro predeterminado.

Cuando se llama a un constructor predeterminado generado por el compilador e intenta utilizar paréntesis, se emite una advertencia:

```

class myclass{};
int main(){
    myclass mc();      // warning C4930: prototyped function not called (was a variable definition intended?)
}

```

Este es un ejemplo del problema de Most Vexing Parse. Dado que la expresión del ejemplo se puede interpretar como la declaración de una función o como la invocación de un constructor predeterminado, y dado que los analizadores de C++ prefieren las declaraciones sobre otras cosas, la expresión se trata como una declaración de función. Para obtener más información, vea [la mayoría de Vexing Parse](#).

Si se declaran constructores no predeterminados, el compilador no proporcionará un constructor predeterminado:

```

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, una matriz de objetos de esa clase no se puede crear únicamente mediante una sintaxis de corchetes. Por ejemplo, dado el bloque de código anterior, no se puede declarar una matriz de marcos como esta:

```
Box boxes[3]; // C2512: no appropriate default constructor available
```

Sin embargo, puede usar un conjunto de listas de inicializadores para inicializar una matriz de objetos de cuadro:

```
Box boxes[3]{ { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

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

Constructores de copias

Un *constructor de copias* inicializa un objeto mediante la copia de los valores de miembro de un objeto del mismo tipo. Si los miembros de clase son tipos simples, como valores escalares, el constructor de copias generado por el compilador es suficiente y no es necesario definir los suyos propios. Si la clase requiere una inicialización más compleja, debe implementar un constructor de copias personalizado. Por ejemplo, si un miembro de clase es un puntero, debe definir un constructor de copias para asignar una nueva memoria y copiar los valores del objeto señalado en el otro. El constructor de copias generado por el compilador simplemente copia el puntero, de modo que el nuevo puntero siga señalando a la ubicación de la memoria del otro.

Un constructor de copias puede tener una de estas firmas:

```

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 copias, también debe definir un operador de asignación de copia (=). Para obtener más información, vea constructores de [asignación](#) y de [copia y operadores de asignación de copia](#).

Puede evitar que se copie el objeto definiendo el constructor de copias como eliminado:

```
Box (const Box& other) = delete;
```

Al intentar copiar el objeto se produce el error *C2280: intentando 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 adicionales deben tener valores predeterminados. Los constructores de movimiento pueden aumentar significativamente la eficacia del programa al pasar objetos de gran tamaño.

```
Box(Box&& other);
```

El compilador elige un constructor de movimiento en determinadas situaciones en las que el objeto se inicializa con otro objeto del mismo tipo que 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 *xValue* (valor que expira). No se asigna a ninguna variable y, por lo tanto, está fuera del ámbito. Para proporcionar motivación a este ejemplo, vamos a dar a `Box` un vector grande de cadenas que represente su contenido. En lugar de copiar el vector y sus cadenas, el constructor de movimiento "lo robará del valor de expiración" `box` "para que el vector pertenezca ahora al nuevo objeto. La llamada a `std::move` es todo lo que se necesita porque `vector` `string` las clases y implementan sus propios constructores de movimiento.

```

#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 copias declarado por el usuario, operador de asignación de copia, operador de asignación de movimiento o destructor. Si no se define ningún constructor de movimiento explícito o implícito, las operaciones

que, de otro modo, usarán un constructor de movimiento, utilizarán en su lugar el constructor de copias. Si una clase declara un constructor de movimiento o un operador de asignación de movimiento, el constructor de copias declarado implícitamente se define como eliminado.

Un constructor de movimiento declarado implícitamente se define como Deleted si alguno de los miembros que son tipos de clase carece de un destructor o el compilador no puede determinar qué constructor se debe usar para la operación de movimiento.

Para obtener más información sobre cómo escribir un constructor de movimiento no trivial, vea constructores de movimiento y operadores de asignación de movimiento (C++).

Constructores explícitamente 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.

```
class Box
{
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, vea [funciones predeterminadas y eliminadas explícitamente](#).

constructores constexpr

Un constructor se puede declarar como `constexpr` si

- se declara como valor predeterminado o bien 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 un bloque try de función;
- se inicializan 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 la lista de inicializadores

Si un constructor toma un `STD::initializer_list <T>` como su parámetro y cualquier otro parámetro tiene argumentos predeterminados, ese constructor se seleccionará en la resolución de sobrecarga cuando se cree una instancia de la clase a través de la inicialización directa. Puede utilizar la `initializer_list` para inicializar cualquier miembro que pueda aceptarla. Por ejemplo, supongamos que la clase `Box` (mostrada anteriormente) tiene un `std::vector<string>` miembro `m_contents`. Puede proporcionar un constructor similar al siguiente:

```
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 de cuadro como este:

```
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:

```
Box(int size): m_width(size), m_length(size), m_height(size){}
```

Se puede inicializar un objeto `Box` como este:

```
Box b = 42;
```

O pasar un valor `int` a una función que toma un objeto `Box`:

```
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 norma general, debe utilizar la `explicit` palabra clave en un constructor (y los operadores definidos por el usuario) para evitar este tipo de conversión implícita de tipos:

```
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, vea [conversiones de tipos definidos por el usuario](#).

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. Primero, se llama al constructor base, después los miembros de la clase base se inicializan en el orden en que aparecen en la declaración de clase y después se llama al constructor derivado.

```
#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:

```
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. Se llama a los constructores de clase base en orden de derivación; por ejemplo, si `ClassA` se deriva de `ClassB`, que se deriva de `ClassC`, `ClassC` se llama primero al constructor, después el `ClassB` constructor y, por último, el constructor `ClassA`.

Si una clase base no tiene un constructor predeterminado, debe proporcionar los parámetros de constructor de

clase base en el constructor de clase derivada:

```
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 implementados. Sin embargo, dado que el objeto en sí no está totalmente implementado, el destructor no se ejecutará.

Constructores derivados e inicialización de agregados extendidos

Si el constructor de una clase base no es público, pero accesible para una clase derivada, en el modo /STD: c++ 17 en Visual Studio 2017 y versiones posteriores, no se pueden usar llaves vacías para inicializar un objeto del tipo derivado.

En el ejemplo siguiente se muestra el comportamiento correspondiente de C++14:

```
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, el constructor privado `Base` se llamaba a través del 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 :

```
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': cannot access
               // private member declared in class 'Base'
```

Constructores para las 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 que se muestran en la declaración de la clase derivada:

```
#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:

```
BaseClass1 ctor  
BaseClass2 ctor  
BaseClass3 ctor  
DerivedClass ctor
```

Delegar constructores

Un *constructor de delegación* llama a un constructor diferente en la misma clase para realizar parte del trabajo de inicialización. Esto resulta útil si tiene varios constructores que todos tienen para realizar un trabajo similar. Puede escribir la lógica principal en un constructor e invocarla desde otras personas. En el siguiente ejemplo trivial, Box (int) delega su cuadro de trabajo a (int, int, int):

```
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, vea [delegar constructores](#).

Heredar constructores (C++ 11)

Una clase derivada puede heredar los constructores de una clase base directa mediante una `using` declaración, tal como se muestra en el ejemplo siguiente:

```

#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 `using` instrucción en el modo `/STD: c++ 17` pone en el ámbito todos los constructores de la clase base, excepto aquellos que tienen una firma idéntica a los constructores de la clase derivada. En general, es mejor usar constructores que heredan cuando la clase derivada no declara ningún constructor o 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:

```

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`:

```
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](#)

Consulta también

[Clases y estructuras](#)

Constructores de copia y operadores de asignación de copia (C++)

06/03/2021 • 6 minutes to read • [Edit Online](#)

NOTE

A partir de C++ 11, se admiten dos tipos de asignación en el idioma: *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, vea [constructores de movimiento y operadores de asignación 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,

```
Point a, b;  
...  
a = b;
```

hace que el valor de `b` se copie en `a`.

- **Inicialización:** la inicialización se produce cuando se declara un nuevo objeto, cuando los argumentos se pasan a funciones por valor o cuando los valores se devuelven de las funciones por valor.

Puede definir la semántica de "copy" para los objetos de tipo de clase. Por ejemplo, considere este código:

```
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 asociar la semántica de copia correspondiente a cada clase, de la manera siguiente.

- Mediante el operador de asignación **Operator =** junto con una referencia al tipo de clase como el tipo de valor devuelto y el parámetro que se pasa por `const` referencia, por ejemplo
`ClassName& operator=(const ClassName& x);`
- Utilizando el constructor de copias.

Si no declara un constructor de copias, el compilador genera uno automáticamente miembro a miembro. 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 se implementa cualquiera de ellos, es recomendable implementar también el otro de modo que el significado del código esté claro.

El constructor de copias toma un argumento de tipo `Class-Name &`, donde `Class-Name` es el nombre de la clase para la que se define el constructor. Por ejemplo:

```
// spec1_copying_class_objects.cpp
class Window
{
public:
    Window( const Window& ); // Declare copy constructor.
    // ...
};

int main()
{
}
```

NOTE

Cree el tipo de la clase de argumento del constructor de copias `const &` siempre que sea posible. Esto evita que el constructor de copias modifique accidentalmente el objeto del que se está copiando. También permite copiar desde `const` objetos.

Constructores de copias generados por el compilador

Los constructores de copias generados por el compilador, como los constructores de copias definidos por el usuario, tienen un único argumento de tipo "referencia a *Class-Name*". Una excepción es cuando todas las clases base y las clases de miembro tienen constructores de copias declarados que toman un solo argumento de tipo `const Class-Name &`. En tal caso, el argumento del constructor de copias generado por el compilador también es `const`.

Cuando el tipo de argumento para el constructor de copias no es `const`, la inicialización mediante la copia de un `const` objeto genera un error. Lo contrario no es cierto: Si el argumento es `const`, puede inicializar mediante la copia de un objeto que no es `const`.

Los operadores de asignación generados por el compilador siguen el mismo patrón con respecto a `const`. Toman un solo argumento de tipo `Class-Name &` a menos que los operadores de asignación de todas las clases base y miembro tomen argumentos de tipo `const Class-Name &`. En este caso, el operador de asignación generado de la clase toma un `const` argumento.

NOTE

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 de argumento no es `const`, la asignación de un `const` objeto genera un error. Lo contrario no es cierto: Si `const` se asigna un valor a un valor que no es `const`, la asignación se realiza correctamente.

Para obtener más información sobre los operadores de asignación sobrecargados, vea [asignación](#).

Constructores de movimiento y operadores de asignación de movimiento (C++)

06/03/2021 • 9 minutes to read • [Edit Online](#)

En este tema se describe cómo escribir un *constructor de movimiento* y un operador de asignación 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 valor l sin copiar. Para obtener más información sobre la semántica de movimiento, vea [declarador de referencia de valor r: &&](#).

Este tema se basa en la siguiente clase de C++, `MemoryBlock`, que administra un búfer de memoria.

```
// 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 (_data != nullptr)
            delete[] _data;
        _length = other._length;
        _data = new int[_length];
        std::copy(other._data, other._data + _length, _data);
    }
}
```

```

        << 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++

- 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:

```

MemoryBlock(MemoryBlock&& other)
: _data(nullptr)
, _length(0)
{
}

```

- En el constructor de movimiento, asigne los miembros de datos de clase del objeto de origen al objeto que se está construyendo:

```

_data = other._data;
_length = other._length;

```

- 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):

```

other._data = nullptr;
other._length = 0;

```

Para crear un operador de asignaciones de movimiento para una clase de C++

- 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:

```

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.

```
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:

```
// 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:

```
// 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:

```
return *this;
```

Ejemplo: completar el constructor de movimiento y el operador de asignación

En el ejemplo siguiente se muestra el constructor de movimiento y el operador de asignación de movimiento completos para la clase `MemoryBlock`:

```

// 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 de 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. La `vector` clase usa la semántica de movimiento para realizar la operación de inserción eficazmente moviendo los elementos del vector en lugar de copiarlos.

```

// 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:

```

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 genera el siguiente resultado:

```

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:

```
// Move constructor.  
MemoryBlock(MemoryBlock&& other) noexcept  
: _data(nullptr)  
, _length(0)  
{  
    *this = std::move(other);  
}
```

La función [STD:: Move](#) convierte el valor l `other` en un valor r.

Consulta también

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

[STD:: Move](#)

Constructores de delegación

15/04/2020 • 3 minutes to read • [Edit Online](#)

Muchas clases tienen varios constructores que realizan acciones similares, por ejemplo, validan parámetros:

```
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 (. . .)`:

```
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 miembro en un constructor que delega en otro constructor,

tal como se muestra aquí:

```
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:

```
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.

```
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++)

06/03/2021 • 13 minutes to read • [Edit Online](#)

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 proporcionará uno predeterminado; para muchas clases es suficiente. Solo necesita 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`:

```
// 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 `delete` operador para desasignar el espacio asignado dinámicamente para el almacenamiento de texto.

Declarar 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 aceptan argumentos.

- No devuelva un valor (o `void`).
- No se puede declarar como `const`, `volatile` o `static`. Sin embargo, se pueden invocar para la destrucción de objetos declarados como `const`, `volatile` o `static`.
- Se puede declarar como `virtual`. Mediante los destructores virtuales, puede destruir objetos sin conocer su tipo; se invoca el destructor correcto para el objeto mediante el mecanismo de función virtual. Observe que los destructores también se pueden declarar como funciones virtuales puras para las 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 `new` operador se desasigna explícitamente mediante `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 se 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 de inicialización de miembros opcional utilizada 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 a la declaración.
4. Los destructores para las clases base virtuales se llaman en el orden inverso al de la declaración.

```

// 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.

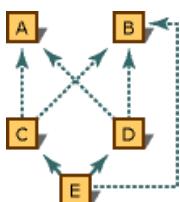


Gráfico de herencia que muestra clases base virtuales

A continuación se enumeran los encabezados de las clases que se muestran en la ilustración.

```

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, omitirla.
5. Si el nodo recordado no está aún en la lista, agregarla al final 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, desde la última hasta la primera.

El orden de construcción o de destrucción es importante sobre todo cuando los constructores o destructores de una clase se basan en que el otro componente se cree primero o persista más tiempo; por ejemplo, si el destructor para **A** (en la ilustración anterior) se basa en que **B** continúa estando presente cuando se ejecute el 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 las clases base no virtuales se llaman en el orden inverso en el que se declaran los nombres de clase base. Considere la siguiente declaración de clase:

```
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. Normalmente, estos objetos se asignan mediante un operador definido por el usuario **new** que toma un argumento de colocación. El **delete** operador no puede desasignar esta memoria porque no está asignada desde el almacén gratuito (para obtener más información, vea [los operadores New y DELETE](#)). 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:

```
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, es probable que tenga que implementar un constructor de copias y una asignación de copia.

Si el usuario no define estas funciones especiales, el compilador las define de forma implícita. Los operadores de asignación y constructores generados implícitamente realizan una copia miembro a miembro superficial, que es casi seguro si un objeto está administrando un recurso.

En el ejemplo siguiente, el constructor de copias generado implícitamente hará que los punteros `str1.text` y `str2.text` hagan referencia a la misma memoria y, cuando se devuelva desde `copy_strings()`, esa memoria se eliminará dos veces, lo que es un comportamiento indefinido:

```
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 copias o un operador de asignación de copia evita la definición implícita del constructor de movimiento y del operador de asignación de movimiento. En este caso, no proporcionar operaciones de movimiento es normalmente, si la copia es costosa, una oportunidad de optimización perdida.

Consulta también

[Constructores de copia y operadores de asignación de copia](#)

[Constructores de movimiento y operadores de asignación de movimiento](#)

Información general de Funciones miembro

06/03/2021 • 3 minutes to read • [Edit Online](#)

Las funciones miembro son estáticas o no estáticas. El comportamiento de las funciones miembro estáticas difiere de otras funciones miembro porque las funciones miembro estáticas no tienen ningún `this` argumento implícito. Las funciones miembro no estáticas tienen un `this` puntero. 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 las declaraciones de clase ya se tratan como funciones insertadas, puede usar la `inline` palabra clave para documentar el código.

A continuación se muestra un ejemplo de declaración de una función dentro de una declaración de clase:

```
// 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 la definición de una 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:

```
// 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()
{
```

NOTE

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)

06/03/2021 • 2 minutes to read • [Edit Online](#)

La palabra clave **virtual** solo se puede aplicar a funciones 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 obtener más información, vea [funciones virtuales](#).

override (especificador)

06/03/2021 • 2 minutes to read • [Edit Online](#)

Puede usar la palabra clave **override** para designar las funciones miembro que invalidan una función virtual en una clase base.

Sintaxis

```
function-declaration override;
```

Observaciones

la **invalidación** 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 **invalidaciones** para ayudar a evitar el comportamiento inadvertido de herencia en el código. En el ejemplo siguiente se muestra dónde, sin usar **override**, es posible que el comportamiento de la función miembro de la clase derivada no se haya diseñado. El compilador no genera ningún error para este código.

```
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
};
```

Cuando se utiliza **override**, el compilador genera errores en lugar de crear de forma silenciosa nuevas funciones miembro.

```
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 se pueden invalidar y que las clases no se pueden heredar, use la palabra clave [final](#).

Consulta también

[último \(especificador\)](#)

[Palabras clave](#)

final (especificador)

06/03/2021 • 2 minutes to read • [Edit Online](#)

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
```

Observaciones

final es contextual y tiene un significado especial solo cuando se utiliza después de una declaración de función o un nombre de clase. De lo contrario, no es una palabra clave reservada.

Cuando se usa **final** en las 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.

```
class BaseClass
{
    virtual void func() final;
};

class DerivedClass: public BaseClass
{
    virtual void func(); // compiler error: attempting to
                        // override a final function
};
```

Para obtener información sobre cómo especificar que se pueden invalidar las funciones miembro, vea [especificador de invalidación](#).

En el ejemplo siguiente se usa la palabra clave **final** para especificar que una clase no se puede heredar.

```
class BaseClass final
{
};

class DerivedClass: public BaseClass // compiler error: BaseClass is
                                    // marked as non-inheritable
{
```

Consulta también

[Palabras clave](#)

`override` (especificador)

Herencia (C++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

En esta sección se explica cómo utilizar clases derivadas para crear programas extensibles.

Introducción

Las clases nuevas pueden derivarse de clases existentes mediante un mecanismo denominado "herencia" (vea la información que comienza en una [herencia única](#)). Las clases que se utilizan para la derivación se conocen como "clases base" de una clase derivada determinada. Una clase derivada se declara mediante la sintaxis siguiente:

```
class Derived : [virtual] [access-specifier] Base
{
    // member list
};
class Derived : [virtual] [access-specifier] Base1,
    [virtual] [access-specifier] Base2, . . .
{
    // member list
};
```

Detrás de la etiqueta (nombre) de la clase, aparece un carácter de dos puntos seguido de una lista de especificaciones bases. Las clases base designadas de esta forma deben haberse declarado previamente. Las especificaciones base pueden contener un especificador de acceso, que es una de las palabras clave `public`, `protected` o `private`. Estos especificadores de acceso aparecen delante del nombre de la clase base y solo se aplican a esa clase base. Estos especificadores controlan el permiso de la clase derivada para su uso en miembros de la clase base. Vea [Access Control de miembro](#) para obtener información sobre el acceso a miembros de clase base. Si se omite el especificador de acceso, se considera el acceso a esa base `private`. Las especificaciones base pueden contener la palabra clave `virtual` para indicar la herencia virtual. Esta palabra clave puede aparecer delante o detrás del especificador de acceso, si hay alguno. Si se usa la herencia virtual, la clase base se conoce como clase base virtual.

Se pueden especificar varias clases base, separadas por comas. Si se especifica una sola clase base, el modelo de herencia es una [herencia única](#). Si se especifica más de una clase base, el modelo de herencia se denomina [herencia múltiple](#).

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` y `_interface` se documentan en esta sección.

Vea también

Funciones virtuales

06/03/2021 • 6 minutes to read • [Edit Online](#)

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.

Supongamos 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

```
// 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:

```

// 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.
}

```

```

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 las funciones virtuales para objetos de tipos de clase, no se pueden declarar funciones globales o estáticas como `virtual`.

La `virtual` palabra clave se puede usar al declarar funciones de reemplazo en una clase derivada, pero no es necesario; las invalidaciones de las funciones virtuales siempre son virtuales.

Las funciones virtuales de una clase base se deben definir a menos que se declaren mediante el *especificador Pure*. (Para obtener más información sobre las funciones virtuales puras, vea [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:

```
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

06/03/2021 • 7 minutes to read • [Edit Online](#)

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.

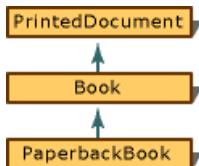


Gráfico sencillo de herencia única

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:

```
// deriv_SingleInheritance.cpp
// compile with: /LD
class PrintedDocument {};

// Book is derived from PrintedDocument.
class Book : public PrintedDocument {};

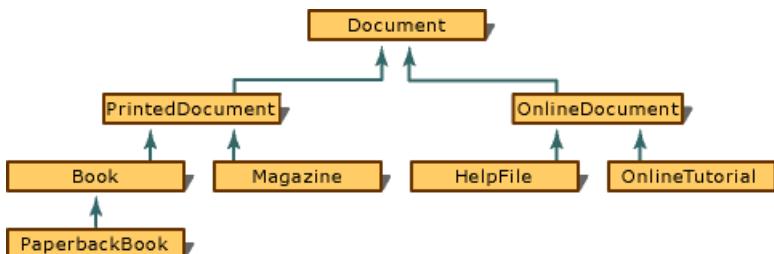
// PaperbackBook is derived from Book.
class PaperbackBook : public Book {};
```

`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 la herencia pública, protegida y privada se describe en [Access Control de miembro](#).

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".

NOTE

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:

```
// 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:

```
// 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 (

::):

```
// 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):

```
// 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.).

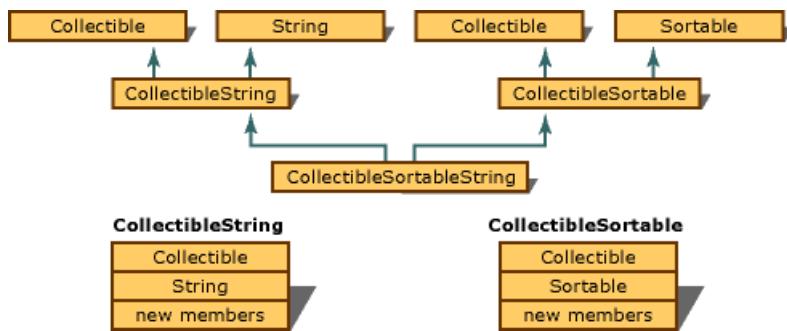
NOTE

Forzar la clase base para implementar una función como `PrintNameOf` no suele ser el mejor diseño. [Funciones virtuales](#) ofrece otras alternativas de diseño.

Clases base

06/03/2021 • 2 minutes to read • [Edit Online](#)

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

06/03/2021 • 15 minutes to read • [Edit Online](#)

Una clase se puede derivar de más de una clase base. En un modelo de herencia múltiple (donde las clases se derivan de más de una clase base), las clases base se especifican mediante el elemento de gramática de la *lista base*. Por ejemplo, se puede especificar la declaración de clase para `CollectionOfBook`, derivada de `Collection` y `Book`:

```
// 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 se realiza 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. Se llama a los destructores en el orden inverso de las clases especificadas en la *lista base*.

NOTE

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.

Al especificar 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.

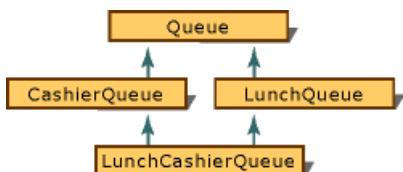
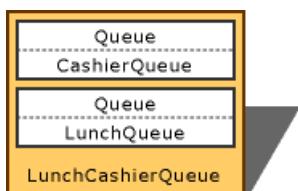


Gráfico de línea de comida simulada

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).

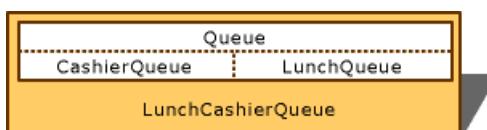


Objeto de línea de comida simulada

Observe que hay dos subobjetos `Queue` en el objeto `LunchCashierQueue`. El código siguiente declara `Queue` como clase base virtual:

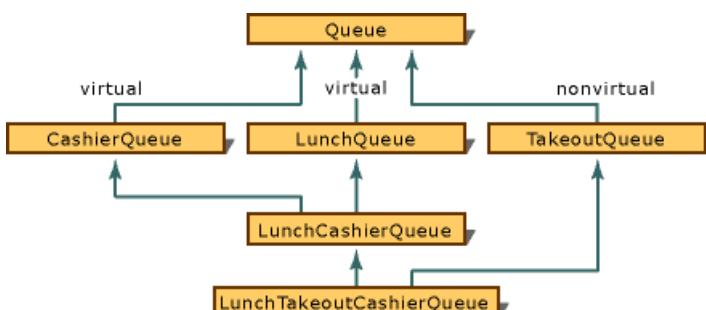
```
// deriv_VirtualBaseClasses.cpp
// compile with: /LD
class Queue {};
class CashierQueue : virtual public Queue {};
class LunchQueue : virtual public Queue {};
class LunchCashierQueue : public LunchQueue, public CashierQueue {};
```

La `virtual` palabra clave garantiza que solo se incluye una copia del subobjeto `Queue` (vea la ilustración siguiente).



Objeto de línea de comida simulada con clases base 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

NOTE

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 de vtordisp. La `/vd1` opción del compilador, que es el valor predeterminado, los habilita cuando es necesario. 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 un módulo de compilación completo. Use la `vtordisp` Directiva pragma para suprimir y, a continuación, volver a habilitar `vtordisp` los campos clase por clase:

```

#pragma vtordisp( off )
class GetReal : virtual public { ... };
\n#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:

```

// 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 **a** en **A** o en **B**:

```
C *pc = new C;  
pc->b();
```

Consideré 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 [member-access control](#)).

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 *pc = new C;  
pc->B::a();
```

NOTE

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:

```

// deriv_Dominance.cpp
// compile with: /LD
class A {
public:
    int a;
};

class B : public virtual A {
public:
    int a();
};

class C : public virtual A {};

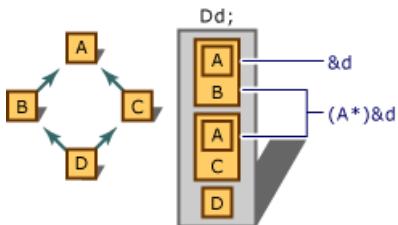
class D : public B, public C {
public:
    D() { a(); } // Not ambiguous. B::a() dominates A::a.
};

```

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 Address-of (`&`) 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:

```

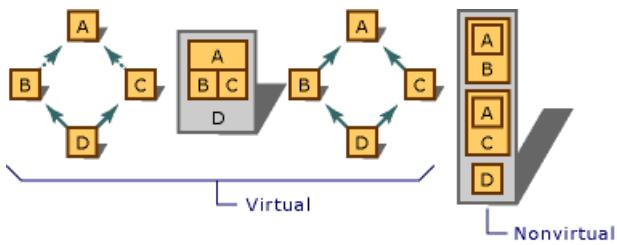
(A*)(B*)&d      // Use B subobject.
(A*)(C*)&d      // Use C subobject.

```

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++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Si la misma función virtual se declara en dos o más [interfaces](#) y si una clase se deriva de estas interfaces, puede invalidar explícitamente cada función virtual.

Para obtener información sobre las invalidaciones explícitas en código administrado con C++/CLI, vea [invalidaciones explícitas](#).

FIN de Específicos de Microsoft

Ejemplo

En el ejemplo de código siguiente se muestra cómo utilizar las invalidaciones explícitas:

```
// 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;
}

```

```

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++)

13/04/2021 • 4 minutes to read • [Edit Online](#)

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. Sin embargo, puede utilizar punteros y referencias a tipos de clase abstractos.

Cree una clase abstracta declarando 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. Esto significa que `Account` es un buen candidato para una clase abstracta:

```
// 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 de 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

Las funciones virtuales puras de las clases abstractas se pueden *definir* o tener una implementación de. Solo puede llamar a estas funciones mediante la sintaxis completa:

`abstract-Class-Name::function-Name()`

Las funciones virtuales puras definidas son útiles cuando se diseñan 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:

```
// 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 cómo una extensión de compilador de Microsoft le permite agregar una definición en línea a un virtual puro `~base()`. También puede definirlo 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 de `base` la clase después del `derived` destructor. La implementación vacía de la función virtual pura `~base` garantiza que exista al menos alguna implementación para la función. Sin él, el vinculador genera un error de símbolo externo sin resolver para la llamada implícita.

NOTE

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 uso de un nombre de función de miembro completo. Dichas funciones deben tener una implementación o la llamada produce un error en el momento de la vinculación.

Consulte también

[Herencia](#)

Resumen de reglas de ámbito

06/03/2021 • 6 minutes to read • [Edit Online](#)

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 sigue siendo inequívoco, 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 miembros para objetos y referencias (.)
- Selección de miembros 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.

Nombres especificados después del operador de selección de miembro (.`o ->`) deben ser miembros del tipo de clase del objeto especificado a la izquierda del operador o de sus clases base. Los nombres especificados a la derecha del operador de selección de miembros (`->`) 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 algún otro tipo de clase. (Esta disposición se describe 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 `class` `struct` `union` palabras clave, y obligan al compilador a buscar solo `class` `struct` los nombres de, o `union`.
3. Los nombres del lado izquierdo del operador de resolución de ámbito (`::`) solo pueden ser `class` nombres,, `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. Del mismo modo, si el nombre hace referencia a cualquier miembro no estático en una clase envolvente, se genera un mensaje de error porque las clases cerradas no tienen punteros de clase de inclusión `this`.

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

16/04/2021 • 3 minutes to read • [Edit Online](#)

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 la clase . Por ejemplo:

```
class S;  
int S::*p;
```

En el código anterior, `p` se declara como un puntero a un miembro entero de la clase S. Sin `class S` embargo, 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 para el compilador:

- En la línea de comandos mediante el `/vmg` modificador
- Uso de `pointers_to_members` la pragma
- Uso de las palabras clave de `__single_inheritance` herencia `__multiple_inheritance` , y `__virtual_inheritance` . Esta técnica controla el modelo de herencia clase por clase.

NOTE

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 , puede afectar negativamente al tamaño y la velocidad del archivo ejecutable resultante. Entre más compleja sea la herencia utilizada por una clase, mayor será el número de bytes necesarios para representar un puntero a un miembro de la clase . Y, cuanto mayor sea el código necesario para interpretar el puntero. La herencia única (o no) es menos compleja y la herencia virtual es la más compleja. Los punteros a miembros que se declaran antes de definir la clase siempre usan la representación más grande y compleja.

Si se cambia el ejemplo anterior a:

```
class __single_inheritance S;  
int S::*p;
```

a continuación, independientemente de las opciones de línea de comandos o pragmas que especifique, los

punteros a los miembros de `class S` usarán la representación más pequeña posible.

NOTE

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.

Por compatibilidad con versiones anteriores, , y son sinónimos de , y a menos que se especifique la opción del compilador Deshabilitar extensiones `_single_inheritance` `_multiple_inheritance` de `_virtual_inheritance` `_single_inheritance` `_multiple_inheritance` `_virtual_inheritance` lenguaje). [/Za](#) (

FIN de Específicos de Microsoft

Vea también

[Palabras clave](#)

virtual (C++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

La `virtual` palabra clave declara una función virtual o una clase base virtual.

Sintaxis

```
virtual [type-specifiers] member-function-declarator  
virtual [access-specifier] base-class-name
```

Parámetros

Type-Specifier

Especifica el tipo de valor devuelto de la función miembro virtual.

declarador de función miembro

Declara una función miembro.

Access: especificador

Define el nivel de acceso a la clase base, `public` `protected` o `private`. Puede aparecer antes o después de la `virtual` palabra clave.

nombre de clase base

Identifica un tipo de clase declarado previamente.

Observaciones

Vea [funciones virtuales](#) para obtener más información.

Vea también las siguientes palabras clave: [Class](#), [Private](#), [Public](#) [Protected](#).

Consulta también

Palabras clave

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();
```

Observaciones

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 dentro del cuerpo de una función miembro.

`__super` no se puede usar con una declaración `using`. Vea [usar declaración](#) para obtener más información.

Con la introducción de [atributos](#) que insertan código, es posible que el código contenga una o varias clases base cuyos nombres no sepá pero que contengan métodos a los que desea llamar.

Ejemplo

```
// 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

Vea también

[Palabras clave](#)

`_interface`

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Una interfaz de Microsoft C++ se puede definir de la siguiente manera:

- 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};
```

Observaciones

Una [clase](#) o [struct](#) de C++ se podría implementar con estas reglas, pero las `_interface` aplica.

Por ejemplo, este es un ejemplo de definición de interfaz:

```
_interface IMyInterface {
    HRESULT CommitX();
    HRESULT get_X(BSTR* pbstrName);
};
```

Para obtener información sobre las interfaces administradas, vea [clase interface](#).

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:

```
virtual HRESULT CommitX() = 0;
```

`_interface` implica el `_declspec` modificador novtable.

Ejemplo

En el ejemplo siguiente se muestra cómo utilizar propiedades declaradas en una interfaz.

```
// 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);
}
```

```
p->int_data = 100  
bstr_data = Testing
```

FIN de Específicos de Microsoft

Vea también

[Palabras clave](#)
[Atributos de interfaz](#)

Funciones miembro especiales

06/03/2021 • 3 minutes to read • [Edit Online](#)

Las *funciones miembro especiales* son funciones miembro de clase (o struct) que, en ciertos casos, el compilador genera automáticamente. Estas funciones son el [constructor predeterminado](#), el [destructor](#), el [constructor de copias](#) 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 implícitamente y definir 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, de la misma manera que si la función no se declarara 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 los casos en los que una llamada a una función miembro especial determinada en una clase no tiene sentido, dadas otras 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*, 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 de modo](#) miembro simple del objeto. La inicialización predeterminada deja todas las variables de miembro en un estado indeterminado.

El destructor predeterminado realiza la destrucción de los miembros del objeto. Solo es virtual si un destructor de clase base es virtual.

Las operaciones de asignación y construcción de copia y movimiento predeterminadas realizan copias de patrón de bits de modo de miembro o movimientos de miembros de datos no estáticos. Las operaciones de movimiento solo se generan cuando no se declaran las operaciones de desplazamiento o de copia. Un constructor de copias predeterminado solo se genera cuando no se declara ningún constructor de copias. 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 ninguno operador de asignación de copia. Se elimina implícitamente si se declara una operación de movimiento.

Vea también

[Referencia del lenguaje C++](#)

Miembros estáticos (C++)

06/03/2021 • 3 minutes to read • [Edit Online](#)

Las clases pueden contener datos de miembro y funciones miembro estáticos. Cuando un miembro de datos se declara como `static`, solo se mantiene 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:

```
// 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:

```
long nBytes = BufferedOutput::bytecount;
```

Para que el miembro estático exista, no es necesario que exista ningún objeto del tipo de clase. También se puede tener acceso a los miembros estáticos mediante la selección de miembro (`. -> operadores and`). Por ejemplo:

```
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 [Access Control de miembro](#). 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 lo tanto, el tipo de `BufferedOutput::bytecount` es `long`.

Consulte también

[Clases y structs](#)

Clases C++ como tipos de valor

06/03/2021 • 6 minutes to read • [Edit Online](#)

Las clases de C++ son tipos de valor predeterminados. Se pueden especificar como tipos de referencia, lo que permite que el comportamiento polimórfico admita la programación orientada a objetos. A veces, los tipos de valor se ven desde la perspectiva de la memoria y el control de diseño, mientras que los tipos de referencia son acerca de las clases base y las funciones virtuales para propósitos 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. En el caso de los tipos de referencia, hace que la clase sea no copiable (deshabilite el constructor de copias y el operador de asignación de copia) y use un destructor virtual, que admite el polimorfismo previsto. Los tipos de valor también son sobre el contenido, que, cuando se copian, siempre proporcionan dos valores independientes que se pueden modificar por separado. Tipos de referencia sobre identidad: ¿Qué tipo de objeto es? Por este motivo, los "tipos de referencia" también se conocen como "tipos polimórficos".

Si realmente desea un tipo de referencia (clase base, funciones virtuales), debe deshabilitar explícitamente la copia, como se muestra en la `MyRefType` clase en el código siguiente.

```
// cl /EHsc /nologo /W4

class MyRefType {
private:
    MyRefType & operator=(const MyRefType &);
    MyRefType(const MyRefType &);

public:
    MyRefType () {}
};

int main()
{
    MyRefType Data1, Data2;
    // ...
    Data1 = Data2;
}
```

Al compilar el código anterior se producirá el siguiente error:

```
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

La sobrecarga de asignación de copia se evita debido a nuevas optimizaciones de copia. Por ejemplo, al insertar una cadena en medio de un vector de cadenas, no habrá ninguna sobrecarga de reasignación de copia, solo un movimiento, incluso si se produce un aumento del propio vector. Esto también se aplica a otras operaciones, por ejemplo, realizar una operación de agregar en dos objetos de gran tamaño. ¿Cómo se habilitan estas optimizaciones de operaciones de valores? En algunos compiladores de C++, el compilador lo habilitará de manera implícita, al igual que el compilador puede generar automáticamente los constructores de copia. Sin embargo, en C++, la clase deberá "participar" para trasladar la asignación y los constructores declarándolos en la definición de clase. Esto se logra mediante el uso de la referencia rvalue de signo de y comercial (&&) en las

declaraciones de función miembro apropiadas y la definición de los métodos de asignación de movimiento y del constructor de movimiento. También debe insertar el código correcto para "robar el Trip" del objeto de origen.

¿Cómo se decide si se necesita el traslado habilitado? Si ya sabe que necesita la construcción de copias habilitada, es probable que desee habilitar el movimiento si puede ser más barato que una copia en profundidad. Sin embargo, si sabe que necesita la compatibilidad con Move, no significa necesariamente que desee que la copia esté habilitada. Este último caso se denominaría "tipo de solo movimiento". Un ejemplo que ya está en la biblioteca estándar es `unique_ptr`. Como nota lateral, el antiguo `auto_ptr` está en desuso y se ha reemplazado con `unique_ptr` precisión debido a la falta de compatibilidad con la semántica de transferencia en la versión anterior de C++.

Mediante el uso de la semántica de movimiento, puede devolver por valor o insertar en el medio. Move es una optimización de la copia. La asignación del montón es necesaria como solución alternativa. Considere el siguiente seudocódigo:

```
#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
```

Habilitar Move para tipos de valor adecuados

En el caso de una clase de valor, donde el movimiento puede ser más barato que una copia en profundidad, habilite la construcción de movimiento y la asignación de movimiento para mayor eficacia. Considere el siguiente seudocódigo:

```
#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/asignación de copia, habilite también la construcción o asignación de movimiento si puede ser más barata 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`.

Consulta también

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

[Aquí está otra vez C++](#)

[Referencia del lenguaje C++](#)

[Biblioteca estándar de C++](#)

Conversiones de tipos definidos por el usuario (C++)

06/03/2021 • 20 minutes to read • [Edit Online](#)

Una *conversión* genera un nuevo valor de algún tipo a partir de un valor de un tipo diferente. Las *conversiones estándar* están integradas en el lenguaje C++ y admiten sus tipos integrados, y puede crear *conversiones definidas por el usuario* para realizar conversiones a, desde o entre tipos definidos por el usuario.

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, vea [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 implementarlos 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, vea [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, vea [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, vea [Access Control miembro](#).

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 conocido de una conversión implícita que puede causar problemas es la conversión a `bool`. Hay muchas razones por las que puede que desee crear un tipo de clase que se pueda usar en un contexto booleano (por ejemplo, para que se pueda usar para controlar una `if` instrucción o un bucle), pero cuando el compilador realice una conversión definida por el usuario a un tipo integrado, el compilador podrá aplicar una conversión estándar adicional después. La intención de esta conversión estándar adicional es permitir cosas como la promoción de `short` a `int`, pero también abre la puerta de las conversiones menos obvias, por ejemplo, de `bool` a `int`, que permite usar el tipo de clase en contextos enteros que nunca se han previsto. Este problema concreto se conoce como el *problema de bool seguro*. Este tipo de problema es donde la `explicit` palabra clave puede ayudar.

La `explicit` palabra clave indica al compilador que la conversión especificada no se puede usar para realizar conversiones implícitas. Si desease la comodidad sintáctica de las conversiones implícitas antes de que `explicit` se introdujera la palabra clave, tenía que aceptar las consecuencias no deseadas que a veces se crearan la conversión implícita o usar las funciones de conversión con nombre menos convenientes como solución alternativa. Ahora, al usar la `explicit` palabra clave, puede crear conversiones convenientes que solo se pueden usar para realizar conversiones explícitas o inicialización directa, y que no conducen al tipo de problemas ejemplificados por el problema de bool seguro.

La `explicit` palabra clave se puede aplicar a los constructores de conversión desde C++ 98 y a las funciones de conversión desde C++ 11. En las secciones siguientes se incluye más información sobre cómo usar la `explicit` palabra clave.

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 del tipo integrado `double` a un tipo definido por el usuario `Money`.

```

#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. Sin embargo, en la segunda llamada a `display_balance`, se necesita una conversión porque el tipo del argumento, `double` con un valor de `49.95`, no es lo que espera la función. La función no puede usar este valor directamente, pero dado que hay una conversión del tipo del argumento (`double`) al tipo del parámetro coincidente —, `Money` un valor temporal de tipo `Money` se construye a partir del argumento y se usa para completar la llamada de función. En la tercera llamada a `display_balance`, observe que el argumento no es un `double`, sino que es un `float` con un valor de, `9.99` y que todavía se puede completar la llamada a la función porque el compilador puede realizar una conversión estándar (en este caso, de `float` a `double`) y, a continuación, realizar la conversión definida por el usuario de `double` a `Money` para completar 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

Al declarar un constructor de conversión como `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.

```

#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 debido a la conversión explícita a `Money`, pero observe que el compilador todavía ha ayudado a completar la conversión insertando una conversión implícita de `float` en `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 del tipo definido por el usuario, `Money`, a un tipo integrado, `double`:

```

#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;
}

```

Tenga en cuenta que la variable miembro `amount` se convierte en privada y que se introduce una función de conversión pública en tipo `double` solo para devolver 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 `<<`. Dado que no se define ningún operador de inserción de secuencia para el tipo definido por el usuario `Money`, pero hay uno para el tipo integrado `double`, el compilador puede utilizar la función de conversión de `Money` a `double` para satisfacer 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 del operador de clase derivada `int` no invalida (o incluso influencia) una función de conversión definida por el usuario del operador de clase base `Short`, aunque 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 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.

```
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.

```
#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, el **operador Double** de la función de conversión se ha hecho explícito 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++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 `mutable` de datos, es válido asignar un valor a este miembro de datos a partir de una `const` función miembro.

Sintaxis

```
mutable member-variable-declaration;
```

Observaciones

Por ejemplo, el código siguiente se compilará sin errores porque se `m_accessCount` ha declarado como `mutable` y, por lo tanto, se puede modificar con `GetFlag` aunque `GetFlag` sea una función miembro `const`.

```
// mutable.cpp
class X
{
public:
    bool GetFlag() const
    {
        m_accessCount++;
        return m_flag;
    }
private:
    bool m_flag;
    mutable int m_accessCount;
};

int main()
{}
```

Consulta también

[Palabras clave](#)

Declaraciones de clase anidadas

06/03/2021 • 7 minutes to read • [Edit Online](#)

Una clase puede declararse dentro del ámbito de otra clase. Esta clase se denomina una "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 anidadas:

```
// 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 dentro de `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`.

NOTE

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:

```
// 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 anidadadas

Las funciones miembro declaradas en clases anidadas se pueden definir en el ámbito del archivo. El ejemplo anterior se podría haber escrito:

```

// 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 utiliza la sintaxis *Qualified-type-name* para declarar el nombre de la función. La declaración:

```
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*, son posibles las construcciones de la forma siguiente:

```

typedef BufferedIO::BufferedInput BIO_INPUT;

int BIO_INPUT::read()

```

La declaración anterior es equivalente a la anterior, pero usa un `typedef` nombre 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:

```

// 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:

```
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

06/03/2021 • 2 minutes to read • [Edit Online](#)

Las clases pueden ser anónimas, es decir, se pueden declarar sin un *identificador*. Esto resulta útil cuando se reemplaza un nombre de clase por un `typedef` nombre, como en el siguiente:

```
typedef struct
{
    unsigned x;
    unsigned y;
} POINT;
```

NOTE

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 algunos código de C, el uso de `typedef` junto con estructuras anónimas es frecuente.

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:

```
struct PTValue
{
    POINT ptLoc;
    union
    {
        int iValue;
        long lValue;
    };
};

PTValue ptv;
```

En el código anterior, `iValue` se puede tener acceso a través del operador de selección de miembro de objeto (.) de la siguiente manera:

```
int i = ptv.iValue;
```

Las clases anónimas están sujetas a determinadas restricciones. (Para obtener más información sobre las uniones anónimas, vea [uniones](#)). Clases anónimas:

- No pueden tener un constructor o un destructor.
- No se puede pasar como argumentos a las funciones (a menos que la comprobación de tipos sea derrotada mediante 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.

```
// 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

06/03/2021 • 7 minutes to read • [Edit Online](#)

Las declaraciones de punteros a miembros son casos especiales de declaraciones de puntero. Se declaran mediante la siguiente secuencia:

$\text{Storage-Class-Specifier}_{\text{OPC}} \text{ CV-calificadores}_{\text{OPT}} \text{ Type-Specifier } \text{MS-Modifier}_{\text{OPC}} \text{ Qualified-Name } ::* \text{ CV-} \text{ calificadores}_{\text{OPC}} \text{ Identifier } p.m.-\text{initializer}_{\text{OPC}} ;$

1. El especificador de declaración:

- Un especificador de clase de almacenamiento opcional.
- Los `const` `volatile` especificadores y 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, vea [Modificadores específicos de Microsoft](#).
- El nombre completo de la clase que contiene los miembros a los que se señala.
- `::` Operador.
- `*` Operador.
- Los `const` `volatile` especificadores y opcionales.
- El identificador que denomina el puntero a miembro.

3. Un inicializador de puntero a miembro opcional:

- `=` Operador.
- `&` Operador.
- Nombre completo de la clase.
- `::` Operador.
- 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 miembro no puede apuntar a un miembro estático de la clase, a un miembro de tipo de referencia o a `void`.

Un puntero a un miembro de una clase difiere de un puntero normal: tiene la información de tipo para el tipo del miembro y para 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.

```

// pointers_to_members1.cpp
class Window
{
public:
    Window();                                // Default constructor.
    Window( int x1, int y1,
            int x2, int y2 );                  // Constructor specifying
                                                // 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 `Window` la clase que es de tipo `char*`. El tipo de `pwCaption` es `char * Window::*`. El siguiente fragmento de código declara punteros a las funciones miembro `SetCaption` y `GetCaption`.

```

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`:

```

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 `.*` y `->*` operadores y (los operadores de puntero a miembro) es que el `.*` operador selecciona los miembros a partir de una referencia de objeto o objeto, mientras que el `->*` operador selecciona los miembros a través de un puntero. Para obtener más información sobre estos operadores, vea [expresiones con operadores de puntero a miembro](#).

El resultado de los operadores de puntero a miembro es el tipo del miembro. En este caso, es `char *`.

El fragmento de código siguiente invoca las funciones miembro `GetCaption` y `SetCaption` mediante punteros a miembros:

```
// 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 instancia del miembro estático. Solo existe una instancia de un miembro estático para todos los objetos de una clase determinada. Esto significa que puede usar los operadores normales dirección-de (&) y desreferencia (*).

Punteros a funciones virtuales y de miembro

La invocación de una función virtual a través de una función de puntero a miembro funciona como si se hubiera llamado directamente a la función. La función correcta se busca en la tabla v 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, vea [funciones virtuales](#)).

El código siguiente muestra cómo invocar una función virtual a través de una función de puntero a miembro:

```
// 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->*bfmPrint)();
    bPtr = &dObject;    // Set pointer to address of dObject.
    (bPtr->*bfmPrint)();
}

// Output:
// Print function for class Base
// Print function for class Derived
```

Puntero this

06/03/2021 • 6 minutes to read • [Edit Online](#)

El `this` puntero es un puntero al que solo se puede tener acceso dentro de las funciones miembro no estáticas de un `class` `struct` tipo, o `union`. Señala al objeto para el que se llama a la función miembro. Las funciones miembro estáticas no tienen un `this` puntero.

Sintaxis

```
this  
this->member-identifier
```

Observaciones

El puntero de un objeto `this` no forma parte del propio objeto. No se refleja en el resultado de una `sizeof` instrucción 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:

```
myDate.setMonth( 3 );
```

se puede interpretar como:

```
setMonth( &myDate, 3 );
```

La dirección del objeto está disponible dentro de la función miembro como `this` puntero. La mayoría de los `this` usos de puntero son implícitos. Aunque no es necesario, es válido utilizar un explícito `this` al hacer referencia a los miembros de `class`. Por ejemplo:

```
void Date::setMonth( int mn )  
{  
    month = mn;           // These three statements  
    this->month = mn;    // are equivalent  
    (*this).month = mn;  
}
```

La expresión `*this` se usa normalmente para devolver el objeto actual desde una función miembro:

```
return *this;
```

El `this` puntero también se usa para protegerse de la autoreferencia:

```
if (&Object != this) {  
// do not execute in cases of self-reference
```

NOTE

Dado que el `this` puntero no es modificable, `this` no se permiten las asignaciones al puntero. Las implementaciones anteriores de C++ permitían la asignación a `this`.

En ocasiones, el `this` puntero se usa directamente, por ejemplo, para manipular los datos struct autoress, donde se requiere la dirección del objeto actual.

Ejemplo

```

// 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();
}

```

```

my buffer
your buffer

```

Tipo del this puntero

El `this` tipo del puntero se puede modificar en la declaración de función mediante `const` las `volatile` palabras clave y. Para declarar una función que tenga cualquiera de estos atributos, agregue las palabras clave después de la lista de argumentos de la función.

Considere un ejemplo:

```
// type_of_this_pointer1.cpp
class Point
{
    unsigned X() const;
};
int main()
{}
```

El código anterior declara una función miembro, `x`, en la que el `this` puntero se trata como un `const` puntero a un `const` objeto. Se pueden utilizar combinaciones de opciones *CV-mod-List*, pero siempre modifican el objeto al que apunta el `this` puntero, no el propio puntero. La declaración siguiente declara la función `x`, donde el `this` puntero es un `const` puntero a un `const` objeto:

```
// type_of_this_pointer2.cpp
class Point
{
    unsigned X() const;
};
int main()
{}
```

La `this` sintaxis siguiente describe el tipo de en una función miembro. La *lista 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 class :

[*VC-Qualifier-List*] * class -Type* *** * const this **

En otras palabras, el `this` puntero siempre es un `const` puntero. No se puede reasignar. Los `const` `volatile` calificadores o utilizados en la declaración de función miembro se aplican a la instancia a la class `this` que apunta el puntero, en el ámbito de esa función.

En la tabla siguiente se explica más detalladamente el funcionamiento de estos modificadores.

Semántica de los this modificadores

MODIFICADOR	SIGNIFICADO
<code>const</code>	No se pueden cambiar los datos de miembro; no se pueden invocar funciones miembro que no sean <code>const</code> .
<code>volatile</code>	Los datos de miembro se cargan desde la memoria cada vez que se accede a ellos. deshabilita ciertas optimizaciones.

Es un error pasar un `const` objeto a una función miembro que no es `const`.

Del mismo modo, también es un error pasar un `volatile` objeto a una función miembro que no es `volatile`.

Las funciones miembro declaradas como `const` no pueden cambiar los datos de miembro: en tales funciones,

el `this` puntero es un puntero a un `const` objeto.

NOTE

With struct Ores y de struct Ores no se pueden declarar como `const` o `volatile`. Sin embargo, se pueden invocar en `const` objetos o `volatile`.

Consulte también

[Palabras clave](#)

Campos de bits de C++

06/03/2021 • 4 minutes to read • [Edit Online](#)

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 de la especificación de *declarador de miembro* de campo de bits es la siguiente:

Sintaxis

declarator: *Constant-Expression*

Observaciones

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 el miembro ocupa en la estructura. Se pueden utilizar campos de bits anónimos (es decir, miembros de campos de bits sin identificador) para llenar.

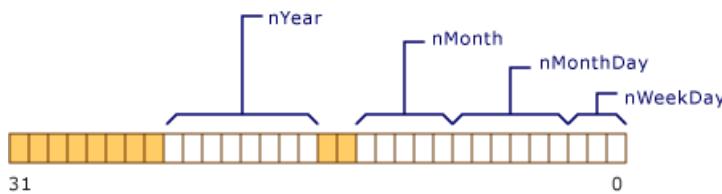
NOTE

Un campo de bits sin nombre de width 0 fuerza la alineación del campo de bits siguiente al siguiente límite de tipo , donde **Type** es el tipo del miembro.

En el ejemplo siguiente se declara una estructura que contiene campos de bits:

```
// bit_fields1.cpp
// compile with: /LD
struct Date {
    unsigned short nYear    : 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 una longitud de 8 bits y desbordaría el límite de palabras del tipo declarado, `unsigned short`. Por lo tanto, se inicia al principio de un nuevo `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

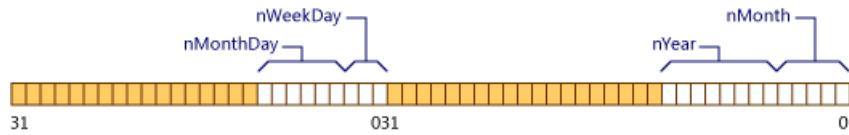
El orden de los datos declarados como campos de bits es de bit bajo a bit alto, como se muestra en la ilustración anterior.

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,

```
// 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)
};
```

después, 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, tal y como se describe en [tipos integrados](#).

Si el inicializador de una referencia de tipo `const T&` es un valor l que hace referencia a un campo de bits de tipo `T`, la referencia no se enlaza directamente al campo de bits. En su lugar, la referencia se enlaza a una inicializada temporal 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++

06/03/2021 • 21 minutes to read • [Edit Online](#)

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 lambdas se usan para encapsular unas líneas de código que se pasan a algoritmos o métodos asíncronos. En este artículo se define qué son las expresiones lambda, se comparan con otras técnicas de programación, se describen sus ventajas y se proporciona un ejemplo básico.

Temas relacionados

- [Expresiones lambda frente a objetos de función](#)
- [Trabajar con expresiones lambda](#)
- [Expresiones lambda constexpr](#)

Partes de una expresión lambda

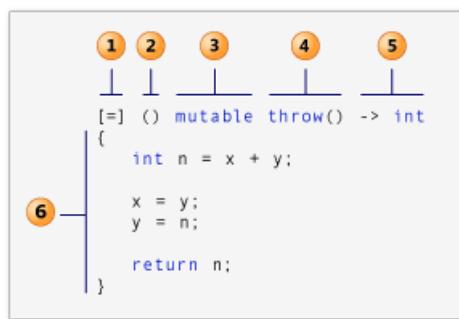
La norma ISO de C++ muestra una expresión lambda sencilla que se pasa como tercer argumento a la función

```
std::sort():
```

```
#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 una expresión lambda:



1. cláusula Capture (también denominada *lambda-presentador* en la especificación de C++).
2. lista de parámetros Opta. (También conocido como *declarador lambda*)
3. especificación mutable Opta.
4. especificación de excepción Opta.
5. finalización: tipo de valor devuelto Opta.

6. cuerpo de la expresión lambda.

Cláusula capture

Una expresión lambda puede introducir nuevas variables en su cuerpo (en C++ 14) y también puede tener acceso a las variables del ámbito circundante, o *capturarlas*. Una expresión lambda comienza con la cláusula Capture (*lambda-presentador* en la sintaxis estándar), que especifica las variables que se capturan y si la captura es por valor o por referencia. A las variables que tienen como prefijo una 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 el modo de captura predeterminado (*captura predeterminada* en la sintaxis estándar) para indicar cómo se capturan las variables externas a las que se hace referencia en la expresión lambda: [&] significa que todas las variables a las que se 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:

```
[&total, factor]
[factor, &total]
[&, factor]
[factor, &]
[=, &total]
[&total, =]
```

Solo se capturan las variables que se mencionan en la expresión lambda cuando se usa una captura de valor predeterminado.

Si una cláusula de captura incluye una captura de forma predeterminada &, no `identifier capture` puede tener el formato en una de la cláusula de captura & `identifier`. Del mismo modo, si la cláusula Capture incluye una captura de forma predeterminada =, no `capture` puede tener la forma = `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.

```
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 paquete, como se muestra en este ejemplo de [plantilla variádicas](#):

```
template<class... Args>
void f(Args... args) {
    auto x = [args...] { return g(args...); };
    x();
}
```

Para usar expresiones lambda en el cuerpo de un método de clase, pase el `this` puntero a la cláusula Capture

para proporcionar acceso a los métodos y miembros de datos de la clase envolvente.

Visual Studio 2017 versión 15,3 y posterior (disponible con [/STD: c++ 17](#)): el `this` puntero se puede capturar por valor especificando `*this` en la cláusula de captura. La captura por valor significa que todo el *cierre*, que es el objeto de función anónimo que encapsula la expresión lambda, se copia en todos los sitios de llamada en los que se invoca la expresión lambda. La captura por valor es útil cuando la expresión lambda se ejecutará en operaciones paralelas o asincrónicas, especialmente en determinadas arquitecturas de hardware, como NUMA.

Para ver un ejemplo en el que se muestra cómo usar expresiones lambda con métodos de clase, vea "ejemplo: usar una expresión lambda en un método" en [ejemplos de expresiones lambda](#).

Al usar la cláusula de captura, se recomienda tener en cuenta estos aspectos importantes, especialmente cuando se usan expresiones lambda con multithreading:

- Se pueden usar capturas por referencia para modificar variables externas, pero no se pueden usar capturas por valor. (`mutable` permite modificar las copias, pero no los 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 una variable local por referencia en una expresión lambda asincrónica, es muy probable que dicha variable desaparezca en cuanto se inicie la expresión lambda, lo que provocará 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 dichas variables existan en el ámbito de inclusión 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. Una ventaja de esta característica es que en C++ 14 puede capturar variables solo de movimiento (por ejemplo, `std::unique_ptr`) del ámbito circundante y usarlas en una expresión lambda.

```
pNums = make_unique<vector<int>>(nums);
//...
auto a = [ptr = move(pNums)]()
{
    // use ptr
};
```

Lista de parámetros

Además de capturar variables, una expresión lambda puede aceptar parámetros de entrada. Una lista de parámetros (*declarador lambda* en la sintaxis estándar) es opcional y, en la mayoría de los aspectos, es similar a la lista de parámetros de una función.

```
auto y = [] (int first, int second)
{
    return first + second;
};
```

En C++ 14, si el tipo de parámetro es genérico, puede usar la `auto` palabra clave 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.

```
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 tema [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 *excepciones-Specification, Finally-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 `mutable` palabra clave cancela esto. No genera miembros de datos mutables. La especificación mutable permite al cuerpo de una expresión lambda modificar las variables que se capturan por valor. En algunos de los ejemplos más adelante en este artículo se muestra cómo usar `mutable`.

Especificación de la excepción

Puede utilizar la `noexcept` especificación de excepción 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 `noexcept` especificación de excepción y el cuerpo de la expresión lambda produce una excepción, como se muestra aquí:

```
// throw_lambda_expression.cpp
// compile with: /W4 /EHsc
int main() // C4297 expected
{
    []() noexcept { throw 5; }();
}
```

Para obtener más información, vea [Especificaciones de excepciones \(Throw\)](#).

Tipo de valor devuelto

El tipo de valor devuelto de una expresión lambda se deduce automáticamente. No tiene que usar la `auto` palabra clave a menos que especifique un *carácter de retorno-return-type*. El *final-return-type* es similar a la parte del tipo de valor devuelto de un método o función normal. Sin embargo, 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 return-type de una expresión lambda si el cuerpo de la expresión lambda contiene una sola instrucción return o si la expresión lambda 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. De lo contrario, el compilador deduce que el tipo de valor devuelto es `void`. Vea los fragmentos de código de ejemplo siguientes que muestran este principio.

```
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 is not 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 de la expresión lambda (*instrucción compuesta* en la sintaxis estándar) de una expresión lambda puede contener todo lo que puede contener el cuerpo de un método o una función ordinarios. 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 clase, cuando se declaran dentro de una clase y `this` se capturan
- Cualquier variable que tenga duración de almacenamiento estática (por ejemplo, las 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:

```
// 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;
}
```

```
5
0
```

Como la variable `n` se captura por valor, el valor sigue siendo `0` después de la llamada a la expresión lambda. La `mutable` especificación permite `n` modificarse dentro de la expresión lambda.

Aunque una expresión lambda solo puede capturar variables que tengan duración de almacenamiento automática, puede utilizar variables que tengan 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.

```
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 is not 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 la biblioteca estándar de C++ `generate_n`. Esta expresión lambda asigna un elemento de un objeto `vector` a la suma de los dos elementos anteriores. La `mutable` palabra clave

se usa para que el cuerpo de la expresión lambda pueda modificar sus copias de las variables externas , que la expresión lambda captura por valor. Como la expresión lambda captura las variables originales por valor, sus valores siguen siendo después de la ejecución de la expresión.

```
// 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 is not 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);
}
```

```
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 posterior (disponible con `/std:c++17`): una expresión lambda se puede declarar como `constexpr` o usar en una expresión constante cuando se permite la inicialización de cada miembro de datos que captura o presenta en una expresión constante.

```
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 `constexpr` función:

```
auto answer = [](int n)
{
    return 32 + n;
};

constexpr int response = answer(10);
```

Si una expresión lambda es implícita o explícita `constexpr`, la conversión a un puntero de función produce una `constexpr` función:

```
auto Increment = [](int n)
{
    return n + 1;
};

constexpr int(*inc)(int) = Increment;
```

Específico de Microsoft

Las expresiones lambda no se admiten en las siguientes entidades administradas 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, `parameter-declaration-clause` por ejemplo:

```
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 sobre él en la sección [Modificadores específicos de Microsoft](#) de la documentación.

Además de la funcionalidad lambda estándar de C++ 11, Visual Studio es compatible con las expresiones lambda sin estado, que se pueden convertir en punteros de función que usan convenciones de llamada arbitrarias.

Vea también

[Referencia del lenguaje C++](#)

[Objetos de función en la biblioteca estándar de C++](#)

[Llamada de función](#)

[for_each](#)

Sintaxis de la expresión lambda

06/03/2021 • 7 minutes to read • [Edit Online](#)

En este artículo se demuestra la sintaxis y los elementos estructurales de las expresiones lambda. Para obtener una descripción de las expresiones lambda, vea [expresiones lambda](#).

Objetos de función frente a expresiones lambda

Al escribir código, es probable que utilice punteros de función y objetos de función para solucionar problemas y realizar cálculos, especialmente cuando se utilizan [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. Como 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

```

// 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;
}

```

```

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 de 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 usa un objeto de función en lugar de una expresión lambda, junto con la función `for_each`, para generar los mismos resultados que 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 de función, `operador ()`. El compilador de Microsoft C++ genera código que es comparable en el tamaño y el rendimiento con el código lambda en el 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 obtener más información sobre el operador () , vea [llamada de función](#). Para obtener más información sobre la función `for_each` , vea [for_each](#).

Código

```
// 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;
}
```

```
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](#)

[crear](#)

[generate_n](#)

[for_each](#)

[Especificaciones de excepciones \(Throw\)](#)

[ADVERTENCIA del compilador \(nivel 1\) C4297](#)

[Modificadores específicos de Microsoft](#)

Ejemplos de expresiones lambda

06/03/2021 • 14 minutes to read • [Edit Online](#)

En este artículo se muestra cómo usar expresiones lambda en programas. Para obtener información general sobre las expresiones lambda, vea [expresiones lambda](#). Para obtener más información sobre la estructura de una expresión lambda, vea [Sintaxis de expresiones lambda](#).

Declarar expresiones lambda

Ejemplo 1

Dado que una expresión lambda tiene tipo, puede asignarla a una `auto` variable o a un `function` objeto, como se muestra aquí:

Código

```
// 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;
}
```

Output

```
5
7
```

Observaciones

Para obtener más información, vea [auto](#), [function](#) clase 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 en lugar de cuando se llama a la expresión. 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.

Código

```
// 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;
}
```

Output

```
47
```

[\[En este artículo\]](#)

Llamar a expresiones lambda

Es posible llamar a una expresión lambda inmediatamente, como se muestra en el fragmento de código siguiente. El segundo fragmento de código 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`:

Código

```
// 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;
}
```

Output

Ejemplo 2

En este ejemplo se pasa una expresión lambda como argumento a la función `find_if`. La expresión lambda devuelve `true` si su parámetro es un número par.

Código

```
// 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;
    }
}
```

Output

```
The first even number in the list is 42.
```

Observaciones

Para obtener más información acerca de la `find_if` función, vea [find_if](#). Para obtener más información acerca de las funciones de la biblioteca estándar de C++ que realizan algoritmos comunes, vea [<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.

Código

```
// 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;
}
```

Output

```
13
```

Observaciones

En este ejemplo, `[](int y) { return y * 2; }` es la expresión lambda anidada.

[\[En este artículo\]](#)

Higher-Order funciones lambda

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 que devuelve una expresión lambda. Puede usar la `function` clase 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.

Código

```

// 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;
}

```

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. Puede capturar explícitamente o implícitamente el `this` puntero para proporcionar acceso a las funciones y miembros de datos de la clase envolvente. **Visual Studio 2017 versión 15,3 y posterior** (disponible con `/std:c++17`): Capture `this` por valor (`[*this]`) cuando la expresión lambda se usará en operaciones asíncronas o paralelas en las que el código pueda ejecutarse después de que el objeto original salga del ámbito.

Puede usar el `this` puntero explícitamente en una función, como se muestra aquí:

```

// 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 puede capturar el `this` puntero implícitamente:

```

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.

```

// 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);
}

```

Output

```
3
6
9
12
```

Observaciones

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 `this` para que pueda tener acceso al `_scale` miembro.

[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 `negate_all` función aplica el unario `operator-` a cada elemento del `vector` objeto. La función `print_all` imprime en la consola todos los elementos del objeto `vector`.

Código

```
// 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);
}
```

Output

```
34
-43
56
After negate_all():
-34
43
-56
```

Observaciones

Para obtener más información acerca de las plantillas de C++, consulte [plantillas](#).

[En este artículo]

Controlar 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 `for_each` función y una expresión lambda para llenar un `vector` objeto con los valores de otro. Utiliza un `try / catch` bloque para controlar el acceso no válido al primer vector.

Código

```
// 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;
    };
}
```

Output

```
Caught 'invalid vector<T> subscript'.
```

Observaciones

Para obtener más información sobre el control de excepciones, vea [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.

Código

```
// 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::ToString(ch));
    }("Hello");
}
```

Output

```
Hello!
```

Observaciones

También se pueden usar expresiones lambda con la biblioteca de STL/CLR. Para obtener más información, vea referencia de la [biblioteca de STL/CLR](#).

IMPORTANT

Las expresiones lambda no se admiten en estas entidades administradas de Common Language Runtime (CLR):

`ref class` , `ref struct` , `value class` y `value struct` .

[En este artículo]

Consulte también

[Expresiones lambda](#)

[Sintaxis de expresiones lambda](#)

`auto`

`function` Las

`find_if`

`<algorithm>`

Llamada de función

Templates (Plantillas [C++])

Control de excepciones

Referencia de la biblioteca de STL/CLR

expresiones lambda de constexpr en C++

06/03/2021 • 2 minutes to read • [Edit Online](#)

Visual Studio 2017 versión 15,3 y posterior (disponible con /STD: c++ 17): una expresión lambda se puede declarar como `constexpr` o usar en una expresión constante cuando se permite la inicialización de cada miembro de datos que captura o presenta en una expresión constante.

```
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 `constexpr` función:

```
auto answer = [](int n)
{
    return 32 + n;
};

constexpr int response = answer(10);
```

Si una expresión lambda es implícita o explícita `constexpr` y se convierte en un puntero de función, la función resultante también será `constexpr`:

```
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 de función](#)

[for_each](#)

Matrices (C++)

06/03/2021 • 18 minutes to read • [Edit Online](#)

Una matriz es una secuencia de objetos del mismo tipo que ocupan un área de memoria contigua. Las matrices de estilo C tradicionales son el origen de muchos errores, pero siguen siendo comunes, especialmente en las bases de código anteriores. En C++ moderno, recomendamos encarecidamente el uso de [STD:: Vector](#) o [STD:: Array](#) en lugar de las matrices de estilo C descritas en esta sección. Ambos tipos de biblioteca estándar almacenan sus elementos como un bloque de memoria contiguo. Sin embargo, proporcionan una mayor seguridad de tipos y admiten iteradores que se garantiza que señalan a una ubicación válida dentro de la secuencia. Para obtener más información, vea [contenedores](#).

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. En el ejemplo siguiente se declara una matriz de 1000 dobles para que se asigne en la pila. El número de elementos se debe proporcionar como un literal entero, o bien como una expresión constante. Esto se debe a que el compilador tiene que saber cuánto espacio de pila se va a asignar; no puede utilizar un valor calculado en tiempo de ejecución. A cada elemento de la matriz se le asigna un valor predeterminado de 0. Si no asigna un valor predeterminado, cada elemento contiene inicialmente los valores aleatorios que se encuentren en esa ubicación de memoria.

```
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 inicial. El último elemento es el elemento ($n - 1$), donde n es el número de elementos que puede contener la matriz. El número de elementos de la declaración debe ser de un tipo entero y debe ser mayor que 0. Es responsabilidad suya asegurarse de que el programa nunca pase un valor al operador de subíndice que sea mayor que `(size - 1)`.

Una matriz de tamaño cero solo es válida cuando la matriz es el último campo de `struct` o `union` y cuando se habilitan las extensiones de Microsoft (`/za` o no se `/permissive-` establece).

Las matrices basadas en la pila son más rápidas para asignar y obtener acceso a las matrices basadas en montones. Sin embargo, el espacio de pila es limitado. El número de elementos de la matriz no puede ser tan grande que use demasiada memoria de la pila. La cantidad de tiempo depende del programa. Puede usar las herramientas de generación de perfiles para determinar si una matriz es demasiado grande.

Declaraciones de montones

Es posible que necesite una matriz demasiado grande para asignarla en la pila o cuyo tamaño no se conozca en tiempo de compilación. Es posible asignar esta matriz en el montón mediante una `new[]` expresión. 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 la pila. También puede usar [aritmética de puntero](#) para pasar el puntero a cualquier elemento arbitrario de la matriz. Es su responsabilidad asegurarse de que:

- siempre se conserva una copia de la dirección del puntero original para que se pueda eliminar la memoria cuando ya no se necesite la matriz.
- no incremente ni disminuya la dirección del puntero después 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 la matriz mediante el operador de subíndice y mediante aritmética de puntero:

```

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

Puede inicializar una matriz en un bucle, un elemento a la vez o en una única instrucción. El contenido de las dos matrices siguientes es idéntico:

```
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 una matriz se pasa a una función, se pasa como un puntero al primer elemento, ya sea una matriz basada en pila o en un montón. El puntero no contiene información de tamaño o tipo adicional. Este comportamiento se denomina *decadencia del puntero*. Al pasar una matriz a una función, siempre debe especificar el número de elementos en un parámetro independiente. Este comportamiento también implica que los elementos de la matriz no se copian cuando la matriz se pasa a una función. Para evitar que la función modifique los elementos, especifique el parámetro como un puntero a `const` los elementos.

En el ejemplo siguiente se muestra una función que acepta una matriz y una longitud. El puntero apunta a la matriz original, no a una copia. Dado que el parámetro no es `const`, la función puede modificar los elementos de la matriz.

```
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 en el bloque de función:

```
void process(const double *p, const size_t len);
```

También se puede declarar la misma función de estas maneras, sin ningún cambio de comportamiento. La matriz se sigue pasando como un puntero al primer elemento:

```
// 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:

```
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

Diseño conceptual de una matriz multidimensional

Puede declarar matrices multidimensionales que tengan 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:

```
// 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 n-1 dimensional. 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 puede omitir la especificación de los límites de la primera dimensión de una matriz multidimensional en las declaraciones de función, como se muestra aquí:

```

// 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;
}

```

The minimum cost to Market 3 is: 17.29

La función `FindMinToMkt` se escribe de forma que agregar nuevos generadores no requiere ningún cambio en el código, solo una nueva compilación.

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 los elementos de la matriz, se utiliza el constructor predeterminado para los elementos restantes. Si no se ha definido 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 (independientemente de si `const`) se pueden inicializar en sus definiciones (fuera de la declaración de clase). Por ejemplo:

```

// 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()
{
}

```

Obtener 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.

```

// 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.

```

// 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 `p2multi` puntero 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 común especificar todos los subíndices, como en la `cout` instrucción, a veces es útil seleccionar un subconjunto específico de elementos de matriz, como se muestra en las instrucciones siguientes `cout`.

Sobrecargar el operador de subíndice

Al igual que otros operadores, el operador de subíndice (`[]`) lo puede redefinir 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 todas las adiciones que implican tipos de puntero, el escalado se realiza automáticamente para ajustarse 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 obtener más información sobre esta conversión, vea [operadores de adición](#).

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`, `Address-of` (`&`) o en la inicialización de una referencia, se convierte en un puntero al primer elemento de la matriz. Por ejemplo:

```

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 modificables. Esta es la razón por la que la asignación siguiente no es válida:

```
szError1 = psz;
```

Vea también

[STD:: Array](#)

Referencias (C++)

06/03/2021 • 3 minutes to read • [Edit Online](#)

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 de valor l que hacen referencia a una variable con nombre y referencias rvalue que hacen referencia a un [objeto temporal](#). El operador & denota una referencia lvalue y el operador && significa una referencia rvalue o una referencia universal (rvalue o lvalue) en función del contexto.

Las referencias se pueden declarar con la sintaxis siguiente:

```
[Storage-Class-Specifier] [ CV-calificadores] tipo-especificadores [ MS-modificador] expresión declaradora [ = ];
```

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-Specifier] [ CV-calificadores] Type-Specifier [ & o && ] [ CV-calificadores] expresión de identificador [ = ];
```

Las referencias se declaran mediante la siguiente secuencia:

1. Los especificadores de la declaración:

- Un especificador de clase de almacenamiento opcional.
- `const` Calificadores opcionales y/o `volatile`.
- El especificador de tipo: el nombre de un tipo.

2. El declarador:

- Modificador opcional concreto de Microsoft. Para obtener más información, vea [Modificadores específicos de Microsoft](#).
- & Operador u operador && .
- Opcional `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 las matrices y funciones. Para obtener más información, vea [punteros](#).

Varios declaradores e inicializadores pueden aparecer en una lista separada por comas detrás de un único especificador de declaración. Por ejemplo:

```
int &i;  
int &i, &j;
```

Las referencias, punteros y objetos se pueden declarar juntos:

```
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

```
// 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);
}
```

```
3
3
4
4
```

Consulte también

[Argumentos de la función de tipo de referencia](#)

[La función de tipo de referencia devuelve](#)

[Referencias a punteros](#)

Declarador de referencia lvalue: &

06/03/2021 • 2 minutes to read • [Edit Online](#)

Contiene la dirección de un objeto, pero se comporta sintácticamente como un objeto.

Sintaxis

```
type-id & cast-expression
```

Observaciones

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 el `&` *identificador* está precedido por un tipo, como `int` o `char`, el *identificador* se declara como una referencia al tipo. Cuando `&` el *identificador* no está precedido de un tipo, el uso es el 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.

```
// 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;
}
```

```
Bill is 40
```

Consulta también

[Referencias](#)

[Argumentos de la función de tipo de referencia](#)

[La función de tipo de referencia devuelve](#)

[Referencias a punteros](#)

Declarador de referencia de valor r: &&

06/03/2021 • 22 minutes to read • [Edit Online](#)

Mantiene una referencia a una expresión de valor R.

Sintaxis

```
type-id && cast-expression
```

Observaciones

Las referencias de valor R permiten distinguir un valor L de un valor R. Las referencias de valor L y valor R son sintácticamente similares, pero siguen reglas algo distintas. Para obtener más información sobre lvalues y rvalues, vea [lvalues y rvalues](#). Para obtener más información sobre las referencias lvalue, consulte [declarador de referencia lvalue: &](#).

En las secciones siguientes se describe cómo las referencias rvalue admiten la implementación de *semántica de transferencia de movimiento* y *reenvío directo*.

Semántica de transferencia de recursos

Las referencias rvalue admiten la implementación de la *semántica de transferencia de movimiento*, lo 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 movimiento, normalmente se proporciona un *constructor de movimiento* y, opcionalmente, un operador de asignación de movimiento (**operador =**), 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 un constructor de movimiento y cómo utilizarlo en la aplicación, vea [constructores de movimiento y operadores de asignación de movimiento \(C++\)](#).

También puede sobrecargar funciones y operadores normales para aprovechar la semántica de transferencia de recursos. Visual Studio 2010 presenta la semántica de transferencia de objetos en la biblioteca estándar de C++.

Por ejemplo, la clase `string` implementa operaciones que realizan semántica de transferencia de recursos.

Considere el ejemplo siguiente que concatena varias cadenas e imprime el resultado:

```

// 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 temporal `string` (un valor r). **Operator +** no puede anexar una cadena a la otra porque no sabe si las cadenas de origen son lvalues o rvalues. Si las cadenas de origen son ambas valores L, puede que se haga referencia a las mismas en otra parte del programa y, por consiguiente, no se deben modificar. Mediante el uso de referencias rvalue, **Operator +** se puede modificar para tomar rvalues, a los que no se puede hacer referencia en otra parte del programa. Por lo tanto, **Operator +** ahora puede anexar una cadena a otra. Esto puede reducir 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 objeto `vector`, el objeto `vector` debe reasignar memoria para sus elementos y, a continuación, copiar cada elemento en otra ubicación de memoria para dejar espacio al elemento insertado. Cuando una operación de inserción copia un elemento, crea un nuevo elemento, llama al constructor de copias para copiar los datos del elemento anterior en el nuevo elemento y, a continuación, 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 transferencia de datos en la biblioteca estándar de C++ en Visual Studio 2010, vea la [biblioteca estándar de c++](#).

Reenvío directo

El reenvío directo reduce la necesidad de funciones sobrecargadas y ayuda a evitar el problema de reenvío. El *problema de reenvío* puede producirse cuando se escribe una función genérica que toma referencias como sus parámetros y pasa (o *reenvía*) estos parámetros a otra función. Por ejemplo, si la función genérica 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 valor R permiten escribir una versión de una función que acepte argumentos arbitrarios y los reenvíe 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 diferente de `const` las referencias que no son de valor `const` I como sus parámetros.

```

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:

```

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`.

```

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` porque `factory` acepta referencias de valor L que son modificables como parámetros, pero la llamada se realiza usando valores R:

```

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:

```

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 función [STD:: Forward](#) es reenviar los parámetros de la función de generador 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` y `Z`. La función `factory` revisada reenvía sus parámetros (ya sean valores L o valores R) al constructor de clase adecuado.

```

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 adicionales de referencias de valor R

Puede sobrecargar una función para que acepte una referencia de valor L y una referencia de valor R.

Al sobrecargar una función para que tome una referencia de valor `const l` o una referencia de valor `r`, puede escribir código que distinga entre objetos no modificables (lvalues) y valores temporales modificables (rvalues). Puede pasar un objeto a una función que toma una referencia `rvalue` 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.

```

// 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 cannot 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:

```

In f(const MemoryBlock&). This version cannot 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 valor R, 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.

Cuando se escribe una función que acepta una referencia de valor R como parámetro, ese parámetro se trata como un valor L en el cuerpo de la función. El compilador trata una referencia de valor R con nombre como un valor L porque se puede hacer referencia a un objeto con nombre en varias partes de un programa; sería peligroso permitir que varias partes de un programa modifiquen o quitan 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 sobrecarga 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 sobrecarga 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 sobrecarga selecciona la versión de `g` que acepta una referencia de valor R porque `f` devuelve una referencia de valor R.

```
// 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:

```
In g(const MemoryBlock&).
In g(MemoryBlock&&).
```

En este ejemplo, la función `main` pasa un valor R 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 la biblioteca estándar de C++ permite convertir un objeto en una referencia rvalue a ese objeto. Como alternativa, puede usar la `static_cast` palabra clave para convertir un valor l en una referencia de valor r, como se muestra en el ejemplo siguiente:

```
// 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:

```
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.

Es habitual escribir una plantilla de función que pase (o *reenvíe*) sus parámetros a otra función. Es importante saber cómo funciona la deducción de tipos de plantilla para plantillas de función que acepten referencias de valor R.

Si el argumento de función es un valor R, el compilador deduce que el argumento será una referencia de valor R. Por ejemplo, si se pasa una referencia de valor R a un objeto de tipo `x` a una función de plantilla que acepta el tipo `T&&` como su parámetro, la deducción de argumento de plantilla deduce que `T` será `x`. Por consiguiente, el parámetro es de tipo `x&&`. Si el argumento de función es un valor l o `const` lvalue, el compilador deduce que su tipo es una referencia lvalue o una `const` referencia 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`.

```

// 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:

```

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. A continuación, el compilador aplica reglas de contracción de referencias cuando sustituye los argumentos de plantilla deducidos para los tipos de parámetro. 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:

```
print_type_and_value<string&>(string& && t)
```

El compilador usa reglas de contracción de referencias para reducir la firma a lo siguiente:

```
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. El reenvío directo de secciones, que se presentó anteriormente en este tema, describe el reenvío directo con mayor detalle.

Resumen

Las referencias de valor R distinguen los valores L de los valores R. Pueden ayudarle a mejorar el rendimiento de las aplicaciones al eliminar la necesidad de asignaciones de memoria y operaciones de copia innecesarias. También permiten escribir una versión de una función que acepte argumentos arbitrarios y los reenvíe a otra función como si se hubiera llamado directamente a la otra función.

Vea también

[Expresiones con operadores unarios](#)

[Declarador de referencia a un valor L: &](#)

[Lvalues y Rvalues](#)

[Constructores de movimiento y operadores de asignación de movimiento \(C++\)](#)

[Biblioteca estándar de C++](#)

Argumentos de función de tipo de referencia

06/03/2021 • 2 minutes to read • [Edit Online](#)

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`:

```
// 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;
}
```

En el código anterior se muestra cómo se obtiene acceso a los miembros de una estructura pasada por referencia mediante el operador de selección de miembro `(.)` en lugar del operador de selección de miembro de puntero `(->)`.

Aunque los argumentos pasados como tipos de referencia observan la sintaxis de los tipos que no son de puntero, conservan 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:

```
long DateOfYear( const Date& date );
```

Este prototipo garantiza que la función `DateOfYear` no cambiará su argumento.

Cualquier función 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 &`.

Consulta también

[Referencias](#)

Valores devueltos de función de tipo de referencia

06/03/2021 • 4 minutes to read • [Edit Online](#)

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.

Del mismo modo que puede ser más eficaz pasar objetos grandes *a* funciones por referencia, también puede ser 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 describen en [operadores sobrecargados](#).

Ejemplo

Considere el ejemplo [Point](#) :

```

// 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

```

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:

- `extern` Declaración 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.

```
// C4172 means Don't do this!!!
Foo& GetFoo()
{
    Foo f;
    ...
    return f;
} // f is destroyed here
```

El compilador emite 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.

Consulta también

[Referencias](#)

Referencias a punteros

06/03/2021 • 3 minutes to read • [Edit Online](#)

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 utiliza como un puntero normal.

Ejemplo

En este ejemplo de código se muestra la diferencia entre el uso de 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.

```
// 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 BTTree {
    char *szText;
    BTTree *Left;
    BTTree *Right;
};

// Define a pointer to the root of the tree.
BTTree *btRoot = 0;

int Add1( BTTree **Root, char *szToAdd );
int Add2( BTTree*& Root, char *szToAdd );
void PrintTree( BTTree* 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 = 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 );
}
}
```

```
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.
```

Consulta también

[Referencias](#)

Punteros (C++)

15/04/2020 • 2 minutes to read • [Edit Online](#)

Un puntero es una variable que almacena la dirección de memoria de un objeto. Los punteros se utilizan ampliamente en C y C++ para tres propósitos principales:

- para asignar nuevos objetos en el montón,
- para pasar funciones a otras funciones
- para iterar elementos en matrices u otras estructuras de datos.

En la programación de estilo C, los *punteros sin procesar* se utilizan para todos estos escenarios. Sin embargo, los punteros sin procesar son el origen de muchos errores de programación graves. Por lo tanto, se desaconseja encarecidamente su uso, excepto cuando proporcionan un beneficio de rendimiento significativo y no hay ambigüedad en cuanto a qué puntero es el *puntero propietario* que es responsable de eliminar el objeto. C++ moderno proporciona punteros inteligentes para asignar *objetos*, *iteradores* para recorrer estructuras de datos y *expresiones lambda* para pasar funciones. Mediante el uso de estos lenguajes y las instalaciones de biblioteca en lugar de punteros sin procesar, hará que su programa sea más seguro, más fácil de depurar y más sencillo de entender y mantener. Consulte [Punteros inteligentes, Iteradores y expresiones de Lambda](#) para obtener más información.

En esta sección

- [Punteros básicos](#)
- [Punteros const y volátiles](#)
- [nuevos y eliminar operadores](#)
- [Punteros inteligentes](#)
- [Cómo: Crear y utilizar instancias de unique_ptr](#)
- [Cómo: Crear y utilizar instancias de shared_ptr](#)
- [Cómo: Crear y utilizar instancias weak_ptr](#)
- [Cómo: Crear y utilizar instancias de CComPtr y CComQIPtr](#)

Consulte también

[Iterators](#)

[Expresiones lambda](#)

Punteros sin formato (C++)

06/03/2021 • 14 minutes to read • [Edit Online](#)

Un *puntero* es un tipo de variable. Almacena la dirección de un objeto en la memoria y se utiliza para tener acceso a ese objeto. Un *puntero sin formato* es un puntero cuya duración no está controlada por un objeto de encapsulado, como un [puntero inteligente](#). A un puntero sin formato se le puede asignar la dirección de otra variable que no sea de puntero, o 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.

```
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*. Un puntero propietario (o una copia de él) debe usarse para liberar explícitamente el objeto asignado por el montón cuando ya no se necesite. Si no se libera la memoria, se producirá una *fuga de memoria* y la ubicación de memoria no estará disponible para ningún otro programa del equipo. La memoria asignada mediante [new](#) debe liberarse mediante [delete](#) (o [** delete \[\]**](#)). Para obtener más información, vea [new delete operadores y](#).

```
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 aumentar o disminuir para apuntar a otra ubicación en la memoria. Esta operación se denomina *aritmética de punteros*. Se usa en la programación de estilo C para recorrer en iteración los elementos de matrices u otras estructuras de datos. [const](#) No se puede hacer que un puntero señale a una ubicación de memoria diferente y, en ese sentido, es similar a una [referencia](#). Para obtener más información, vea [const y punteros volátiles](#).

```
// 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 los sistemas operativos de 64 bits, un puntero tiene un tamaño de 64 bits. El tamaño del puntero del 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 referencias) se usan mucho en C++ para pasar objetos más grandes a las funciones y desde ellas. Esto se debe a que 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 desee que la función modifique el objeto. En general, las `const` referencias son la manera preferida de pasar objetos a las funciones a menos que el valor del objeto pueda ser `nullptr`.

Los [punteros a funciones](#) permiten pasar funciones a otras funciones y se utilizan para "devoluciones de llamada" en la programación de estilo C. C++ moderno usa [expresiones lambda](#) para este propósito.

Inicialización y acceso a miembros

En el ejemplo siguiente se muestra cómo declarar, inicializar y utilizar un puntero sin formato. Se inicializa mediante `new` para señalar un objeto asignado en el montón, que debe ser explícitamente `delete`. En el ejemplo también se muestran algunos de los peligros asociados a los punteros sin formato. (Recuerde que este ejemplo es la programación de estilo C y no C++ moderno).

```
#include <iostream>
#include <string>

class MyClass
{
public:
    int num;
    std::string name;
    void print() { std::cout << name << ":" << num << std::endl; }
};

// Accepts a MyClass pointer
void func_A(MyClass* mc)
{
    // Modify the object that mc points to.
    // All copies of the pointer will point to
    // the same modified object.
    mc->num = 3;
}

// Accepts a MyClass object
void func_B(MyClass mc)
{
    // mc here is a regular object, not a pointer.
    // Use the "." operator to access members.
    // This statement modifies only the local copy of mc.
    mc.num = 21;
    std::cout << "Local copy of mc:";
    mc.print(); // "Erika, 21"
}

int main()
{
    // Use the * operator to declare a pointer type
    // Use new to allocate and initialize memory
    MyClass* pmc = new MyClass{ 108, "Nick" };

    // Prints the memory address. Usually not what you want.
    std::cout << pmc << std::endl;

    // Copy the pointed-to object by dereferencing the pointer
    // to access the contents of the memory location.
    // mc is a separate object, allocated here on the stack
    MyClass mc = *pmc;

    // Declare a pointer that points to mc using the addressof operator
    MyClass* pcopy = &mc;

    // Use the -> operator to access the object's public members
    pmc->print(); // "Nick, 108"
```

```

// Copy the pointer. Now pmc and pmc2 point to same object!
MyClass* pmc2 = pmc;

// Use copied pointer to modify the original object
pmc2->name = "Erika";
pmc->print(); // "Erika, 108"
pmc2->print(); // "Erika, 108"

// Pass the pointer to a function.
func_A(pmc);
pmc->print(); // "Erika, 3"
pmc2->print(); // "Erika, 3"

// Dereference the pointer and pass a copy
// of the pointed-to object to a function
func_B(*pmc);
pmc->print(); // "Erika, 3" (original not modified by function)

delete(pmc); // don't forget to give memory back to operating system!
// delete(pmc2); //crash! memory location was already deleted
}

```

Aritmética de punteros y matrices

Los punteros y las matrices están estrechamente relacionados. Cuando una matriz se pasa por valor a una función, se pasa como un puntero al primer elemento. En el ejemplo siguiente se muestran las siguientes propiedades importantes de los punteros y las matrices:

- el `sizeof` operador devuelve el tamaño total en bytes de una matriz
- para determinar el número de elementos, divida el número total de bytes por el tamaño de un elemento
- Cuando una matriz se pasa a una función, *decaer* en un tipo de puntero
- el `sizeof` operador cuando se aplica a un puntero devuelve el tamaño del puntero, 4 bytes en x86 u 8 bytes en x64

```

#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 no const punteros para que apunten a otra ubicación de memoria. Los punteros se incrementan y disminuyen mediante `++` los `+= -`= operadores, y `--`. Esta técnica se puede utilizar en matrices y es especialmente útil en búferes de datos sin tipo. Un se `void*` incrementa según el tamaño de un `char` (1 byte). Un puntero con tipo se incrementa según el tamaño del tipo al que señala.

En el ejemplo siguiente se muestra cómo se puede usar la aritmética de puntero para tener acceso a píxeles individuales en un mapa de bits de Windows. Observe el uso de `new` y `delete`, y el operador de desreferencia.

```
#include <Windows.h>
#include <fstream>

using namespace std;

int main()
{
    BITMAPINFOHEADER header;
    header.biHeight = 100; // Multiple of 4 for simplicity.
    header.biWidth = 100;
    header.biBitCount = 24;
    header.biPlanes = 1;
    header.biCompression = BI_RGB;
    header.biSize = sizeof(BITMAPINFOHEADER);

    constexpr int bufferSize = 30000;
    unsigned char* buffer = new unsigned char[bufferSize];

    BITMAPFILEHEADER bf;
    bf.bfType = 0x4D42;
    bf.bfSize = header.biSize + 14 + bufferSize;
    bf.bfReserved1 = 0;
    bf.bfReserved2 = 0;
    bf.bfOffBits = sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFOHEADER); //54

    // Create a gray square with a 2-pixel wide outline.
    unsigned char* begin = &buffer[0];
    unsigned char* end = &buffer[0] + bufferSize;
    unsigned char* p = begin;
    constexpr int pixelWidth = 3;
    constexpr int borderWidth = 2;

    while (p < end)
    {
        // Is top or bottom edge?
        if ((p < begin + header.biWidth * pixelWidth * borderWidth)
            || (p > end - header.biWidth * pixelWidth * borderWidth)
            // Is left or right edge?
            || (p - begin) % (header.biWidth * pixelWidth) < (borderWidth * pixelWidth)
            || (p - begin) % (header.biWidth * pixelWidth) > ((header.biWidth - borderWidth) * pixelWidth))
        {
            *p = 0x0; // Black
        }
        else
        {
            *p = 0xC3; // Gray
        }
        p++; // Increment one byte sizeof(unsigned char).
    }

    ofstream wf(R"(box.bmp)", ios::out | ios::binary);

    wf.write(reinterpret_cast<char*>(&bf), sizeof(bf));
    wf.write(reinterpret_cast<char*>(&header), sizeof(header));
    wf.write(reinterpret_cast<char*>(begin), bufferSize);

    delete[] buffer; // Return memory to the OS.
    wf.close();
}
```

void* punteros

Un puntero a `void` simplemente apunta a una ubicación de memoria sin procesar. A veces es necesario usar `void*` punteros, por ejemplo, al pasar entre código C++ y funciones de C.

Cuando un puntero con tipo se convierte en un `void` puntero, 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 o decremento. Se puede convertir una ubicación de memoria, por ejemplo, de `MyClass*` a `void*` y de nuevo a `MyClass*`. Estas operaciones son intrínsecamente propensas a errores y requieren una gran atención a `void` errores. El C++ moderno desaconseja el uso de `void` punteros en casi todas las circunstancias.

```
//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"

    // 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(p);
}
```

Punteros a funciones

En la programación de estilo C, los punteros de función se utilizan 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++ moderno, las [expresiones lambda](#) proporcionan la misma capacidad con mayor seguridad de tipos y otras ventajas.

Una declaración de puntero de función especifica la firma que debe tener la función señalada:

```
// 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 devuelve un `std::string`. Dependiendo de la función que se pasa a `combine`, antepone o anexa una cadena.

```
#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: *](#)
[address-of \(Operador\): &](#)
[Bienvenido de nuevo a C++](#)

punteros const y volatile

06/03/2021 • 6 minutes to read • [Edit Online](#)

Las palabras clave `const` y `volatile` cambian el modo en que se tratan los punteros. La `const` palabra clave especifica que el puntero no se puede modificar después de la inicialización; el puntero está protegido contra modificaciones después.

La `volatile` palabra clave especifica que el valor asociado al nombre que se indica a continuación puede ser modificado por acciones distintas de las de la aplicación de usuario. Por lo tanto, la `volatile` palabra clave es útil para declarar objetos en memoria compartida a los que pueden tener acceso varios procesos o áreas de datos globales que se usan para la comunicación con rutinas de servicio de interrupción.

Cuando un nombre se declara como `volatile`, el compilador recarga el valor de la memoria cada vez que el programa obtiene acceso a él. 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 apunta el puntero como `const` o `volatile`, utilice una declaración con el formato:

```
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:

```
char * const pchc;
char * volatile pchv;
```

El lenguaje C++ impide las asignaciones que permitan la modificación de un objeto o 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:

```
const char cch = 'A';
char ch = 'B';
```

Dadas las declaraciones anteriores de dos objetos (`cch`, de tipo `const char` y `ch`, de tipo `Char`), las siguientes declaraciones o inicializaciones son válidas:

```
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.

```
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 por la que no `pch2` se permite la declaración.

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`.

```
*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`, obedecen a las mismas reglas.

Los punteros a `const` objetos se utilizan a menudo en declaraciones de función como se indica a continuación:

```
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 sería gratuita para modificar ambos `strDestination` y `strSource` si `strSource` no se hubieran declarado como `const`. La declaración de `strSource` as `const` garantiza que el llamador no se `strSource` puede cambiar por la función a la que se llama.

NOTE

Dado que hay una conversión estándar de `TypeName*` a `const TypeName*`, es legal pasar un argumento de tipo `char *` a `strcpy_s`. Sin embargo, lo contrario no es cierto. No existe ninguna conversión implícita para quitar el `const` atributo de un objeto o puntero.

Un `const` puntero de un tipo determinado se puede asignar a un puntero del mismo tipo. Sin embargo, un puntero que no se `const` puede asignar a un `const` puntero. El código siguiente muestra las asignaciones correctas e incorrectas:

```
// 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.

```
// 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;
}
```

Consulta también

[Punteros Punteros sin formato](#)

Operadores `new` y `delete`

02/11/2020 • 10 minutes to read • [Edit Online](#)

C++ admite la asignación y desasignación dinámica de objetos mediante `new` los `delete` operadores y. Estos operadores asignan memoria para los objetos de un conjunto denominado almacén libre. El `new` operador llama a la función especial `operator new` y el `delete` operador llama a la función especial `operator delete`.

La `new` función de la biblioteca estándar de C++ admite el comportamiento especificado en el estándar de C++, que consiste en iniciar una `std::bad_alloc` excepción si se produce un error en la asignación de memoria. Si todavía desea la versión de no lanzamiento de `new`, vincule el programa con `nothrownew.obj`. Sin embargo, cuando se vincula con `nothrownew.obj`, el valor predeterminado `operator new` de la biblioteca estándar de C++ ya no funciona.

Para obtener una lista de los archivos de biblioteca de la biblioteca en tiempo de ejecución de C y la biblioteca estándar de C++, vea características de la [biblioteca CRT](#).

`new` Operador

El compilador traduce una instrucción como esta a una llamada a la función `operator new`:

```
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, las llamadas repetidas para `operator new` devolver distintos punteros. Si no hay memoria suficiente para la solicitud de asignación, se `operator new` produce una `std::bad_alloc` excepción. O bien, devuelve `nullptr` si se ha vinculado en soporte de no generación `operator new`.

Puede escribir una rutina que intente liberar memoria y reintentar la asignación. Para obtener más información, vea [_set_new_handler](#). Para más información sobre el esquema de recuperación, consulte la sección control de la [memoria insuficiente](#).

Los dos ámbitos de las `operator new` funciones se describen en la tabla siguiente.

Ámbito de `operator new` las funciones

OPERADOR	ÁMBITO
<code>::operator new</code>	Global
<code>nombre de clase*** ::operator new *</code>	Clase

El primer argumento `operator new` debe ser de tipo `size_t`, definido en `<stddef.h>`, y el tipo de valor devuelto siempre es `void*`.

`operator new` Se llama a la función global cuando `new` se usa el operador para asignar objetos de tipos integrados, objetos de tipo de clase que no contienen funciones definidas por el usuario `operator new` y matrices de cualquier tipo. Cuando el `new` operador se usa para asignar objetos de un tipo de clase donde `operator new` se define, `operator new` se llama a la clase.

Una `operator new` función definida para una clase es una función miembro estática (que no puede ser virtual)

que oculta la `operator new` función global para los objetos de ese tipo de clase. Considere el caso en el que `new` se usa para asignar y establecer la memoria en un valor determinado:

```
#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 en `new` se pasa a `Blanks::operator new` como `chInit` argumento. Sin embargo, la `operator new` función global está oculta, lo que hace que el código como el siguiente genere un error:

```
Blanks *SomeBlanks = new Blanks;
```

El compilador admite la matriz `new` de miembros y los `delete` operadores en una declaración de clase. Por ejemplo:

```
class MyClass
{
public:
    void * operator new[] (size_t)
    {
        return 0;
    }
    void operator delete[] (void*)
    {
    }
};

int main()
{
    MyClass *pMyClass = new MyClass[5];
    delete [] pMyClass;
}
```

Controlar la memoria insuficiente

Las pruebas de asignación de memoria con errores se pueden realizar como se muestra aquí:

```
#include <iostream>
using namespace std;
#define BIG_NUMBER 100000000
int main() {
    int *pI = new int[BIG_NUMBER];
    if( pI == 0x0 ) {
        cout << "Insufficient memory" << endl;
        return -1;
    }
}
```

Hay otra manera de controlar las solicitudes de asignación de memoria con error. Escriba una rutina de recuperación personalizada para controlar este tipo de error y, a continuación, registre la función mediante una llamada a la `_set_new_handler` función en tiempo de ejecución.

delete Operador

La memoria que se asigna dinámicamente mediante el `new` operador se puede liberar mediante el `delete` operador. El operador Delete llama a la `operator delete` función, que libera la memoria de nuevo en el Grupo disponible. El uso del `delete` operador también hace que se llame al destructor de clase (si existe).

Hay funciones globales y de ámbito de clase `operator delete`. Solo `operator delete` se puede definir una función para una clase determinada; si se define, oculta la función global `operator delete`. `operator delete` Siempre se llama a la función global para matrices de cualquier tipo.

Función global `operator delete`. Existen dos formas para las `operator delete` funciones globales y de miembro de clase `operator delete`:

```
void operator delete( void * );
void operator delete( void *, size_t );
```

Solo una de las dos formas anteriores puede estar presente para una clase determinada. La primera forma toma un solo argumento de tipo `void *`, que contiene un puntero al objeto que se va a desasignar. La segunda forma, la desasignación de 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 se puede devolver un valor).

La intención del segundo formulario es acelerar la búsqueda de la categoría de tamaño correcto del objeto que se va a eliminar. Esta información no suele almacenarse cerca de la propia asignación y es probable que no esté almacenada en caché. La segunda forma resulta útil cuando `operator delete` se usa una función de una clase base para eliminar un objeto de una clase derivada.

La `operator delete` función es estática, por lo que no puede ser virtual. La `operator delete` función obedece el control de acceso, como se describe en [Access Control de miembro](#).

En el ejemplo siguiente se muestran las funciones y definidas por el usuario `operator new` `operator delete` diseñadas para registrar asignaciones y desasignaciones de memoria:

```

#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 usar para detectar "fugas de memoria", es decir, la memoria que se asigna en el almacén libre pero que nunca se libera. Para detectar pérdidas, `new` `delete` se han redefinido los operadores global y para contar la asignación y desasignación de memoria.

El compilador admite la matriz `new` de miembros y los `delete` operadores en una declaración de clase. Por ejemplo:

```
// 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)

06/03/2021 • 15 minutes to read • [Edit Online](#)

En la programación moderna de C++, la biblioteca estándar incluye *punteros inteligentes*, que se usan para ayudar a garantizar que los programas están libres de memoria y pérdidas de recursos y son seguros para las excepciones.

Usos de los punteros inteligentes

Los punteros inteligentes se definen en el `std` espacio de nombres del `<memory>` archivo de encabezado. Son cruciales para el **RAII** o la *adquisición de recursos* es la expresión de programación de inicialización. 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.

```
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.

IMPORTANT

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 `unique_ptr` puede utilizar un tipo de puntero inteligente 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 `new` expresión 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 a un `new` objeto-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 mediante el 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 tiene acceso 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.

```
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 `get` función miembro para este propósito y `CCComPtr` tienen un `p` miembro de clase 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.

```
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());
}
```

Tipos 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 eficaz; el tamaño es un puntero y admite referencias rvalue para la inserción y recuperación rápidas de las

colecciones de la biblioteca estándar de C++. Archivo de encabezado: `<memory>`. Para obtener más información, vea [Cómo: crear y usar instancias de unique_ptr y unique_ptr clase](#).

- **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 shared_ptr clase](#).

- **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 weak_ptr clase](#).

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, vea [Cómo: crear y usar instancias de CComPtr y CComQIPtr](#).

Clase CComQIPtr

Se parece a `CComPtr`, pero también proporciona la sintaxis simplificada para llamar a `QueryInterface` en objetos COM. Para obtener más información, vea [Cómo: crear y usar instancias de CComPtr y CComQIPtr](#).

Clase CComHeapPtr

Puntero inteligente a objetos que utilizan `CoTaskMemFree` para liberar memoria.

Clase CComGITPtr

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 los punteros inteligentes para los objetos COM, ATL también define punteros inteligentes y colecciones de punteros inteligentes para objetos de C++ sin formato (POCO). En la programación clásica de Windows, estos tipos son alternativas útiles a las colecciones de la biblioteca estándar de C++, especialmente cuando no se requiere portabilidad del código o cuando no se desea mezclar los modelos de programación de la biblioteca estándar de C++ y ATL.

Clase CAutoPtr

Puntero inteligente que exige una propiedad única al transferir la propiedad en la copia. Puede compararse con la clase `std::auto_ptr` en desuso.

Clase CHeapPtr

Puntero inteligente para los objetos asignados mediante la función `malloc` de C.

[Clase `CAutoVectorPtr`](#)

Puntero inteligente para matrices que se asignan mediante `new[]`.

[Clase `CAutoPtrArray`](#)

Clase que encapsula una matriz de elementos `CAutoPtr`.

[Clase `CAutoPtrList`](#)

Clase que encapsula los métodos para manipular una lista de nodos de `CAutoPtr`.

Consulta también

[Punteros](#)

[Referencia del lenguaje C++](#)

[Biblioteca estándar de C++](#)

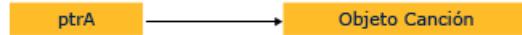
Procedimiento Creación y uso de instancias unique_ptr

06/03/2021 • 4 minutes to read • [Edit Online](#)

Un `unique_ptr` no comparte su puntero. No se puede copiar en otro `unique_ptr`, pasar por valor a una función o usarse en cualquier algoritmo de la biblioteca estándar de C++ que requiera que se realicen 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 lo tanto, si necesita un puntero inteligente para un objeto de C++ sin formato, use `unique_ptr` y, cuando construya un `unique_ptr`, utilice la función auxiliar `make_unique`.

El diagrama siguiente muestra la transferencia de propiedad entre dos instancias de `unique_ptr`.

```
auto ptrA = make_unique<Canción>(L"Diana Krall", L"The Look of Love");
```



```
auto ptrB = std::move(ptrA);
```



`unique_ptr` se define en el `<memory>` encabezado de la biblioteca estándar de C++. Es exactamente tan eficaz como un puntero sin formato y se puede usar en contenedores de la biblioteca estándar de C++. La adición de `unique_ptr` instancias a contenedores de la biblioteca estándar de C++ 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.

```
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.

```
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.

```
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 usar `make_unique` para crear un `unique_ptr` en una matriz, pero no puede usar `make_unique` para inicializar los elementos de la matriz.

```
// 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, vea [make_unique](#).

Consulta también

[Punteros inteligentes \(C++ moderno\)](#)

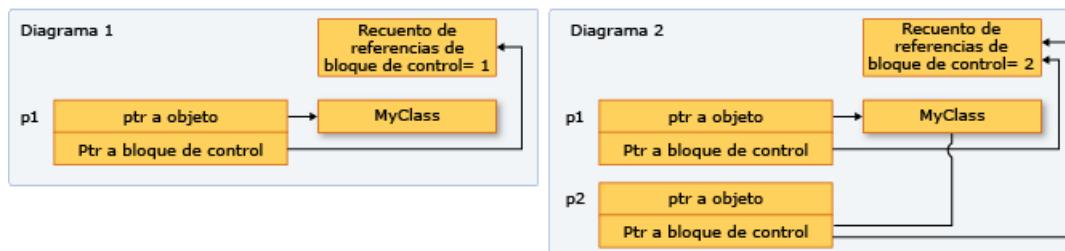
[make_unique](#)

Cómo: crear y usar instancias de shared_ptr

06/03/2021 • 11 minutes to read • [Edit Online](#)

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:

```

// 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 `new` expresión explícita para crear el objeto antes de pasarlo al `shared_ptr` constructor. En el ejemplo siguiente se muestran varias maneras de declarar e inicializar `shared_ptr` junto con un nuevo objeto.

```

// Use make_shared function when possible.
auto sp1 = make_shared<Song>(L"The Beatles", L"Im 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(LLady 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.

```

//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.

```

vector<shared_ptr<Song>> v;

v.push_back(make_shared<Song>(L"Bob Dylan", L"The Times They Are A Changing"));
v.push_back(make_shared<Song>(L"Aretha Franklin", L"Bridge Over Troubled Water"));
v.push_back(make_shared<Song>(L"Thalida", 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 son similares `dynamic_cast` a los `static_cast` operadores, 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.

```
vector<shared_ptr<MediaAsset>> assets;

assets.push_back(shared_ptr<Song>(new Song(L"Himesh Reshammiya", L"Tera Surroor")));
assets.push_back(shared_ptr<Song>(new Song(L"Penaz Masani", L"Tu Dil De De")));
assets.push_back(shared_ptr<Photo>(new 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`.

```
// 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;
```

Consulta también

[Punteros inteligentes \(C++ moderno\)](#)

Procedimiento Creación y uso de instancias weak_ptr

06/03/2021 • 5 minutes to read • [Edit Online](#)

A veces, un objeto debe almacenar una manera de tener acceso al objeto subyacente de una `shared_ptr` sin provocar que se incremente el recuento de referencias. Normalmente, esta situación se produce cuando hay referencias cíclicas entre `shared_ptr` instancias.

El mejor diseño es evitar la propiedad compartida de los punteros siempre que sea posible. Sin embargo, si debe tener la propiedad compartida de `shared_ptr` las instancias, evite las referencias cíclicas entre ellas. Cuando las referencias cíclicas son inevitables, o incluso preferibles por alguna razón, use `weak_ptr` para dar a uno o varios propietarios una referencia débil a otro `shared_ptr`. Mediante el uso de `weak_ptr`, puede crear un `shared_ptr` que se une a un conjunto existente de instancias relacionadas, pero solo si el recurso de memoria subyacente todavía es válido. Una `weak_ptr` propiamente dicha no participa en el recuento de referencias y, por lo tanto, no puede impedir que el recuento de referencias pase a cero. Sin embargo, puede utilizar un `weak_ptr` para intentar obtener una nueva copia de con la `shared_ptr` que se inicializó. Si ya se ha eliminado la memoria, el `weak_ptr` operador bool de 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 `shared_ptr` variable permanezca en el ámbito.

Ejemplo

En el ejemplo de código siguiente se muestra un caso en el que `weak_ptr` se usa para garantizar la eliminación correcta de los objetos que tienen dependencias circulares. Cuando examine el ejemplo, supongamos que se creó solo después de que se tuvieran en cuenta soluciones alternativas. Los `Controller` objetos representan algún aspecto de un proceso de equipo y funcionan de manera independiente. Cada controlador debe ser capaz de consultar el estado de los demás controladores en cualquier momento, y cada uno de ellos contiene un privado `vector<weak_ptr<Controller>>` para este fin. Cada vector contiene una referencia circular y, por lo tanto, `weak_ptr` se usan instancias 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)
    {
        try
        {
            auto p = wp.lock();
            wcout << L"Status of " << p->Num << " = " << p->Status << endl;
        }

        catch (bad_weak_ptr b)
        {
            wcout << L"Null object" << endl;
        }
    });
}

void RunTest()
{
    vector<shared_ptr<Controller>> v;

    v.push_back(shared_ptr<Controller>(new Controller(0)));
    v.push_back(shared_ptr<Controller>(new Controller(1)));
    v.push_back(shared_ptr<Controller>(new Controller(2)));
    v.push_back(shared_ptr<Controller>(new Controller(3)));
    v.push_back(shared_ptr<Controller>(new 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);
}

```

```
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 un `vector<shared_ptr<Controller>>` y, a continuación, en la salida, observe que no se invoca ningún destructor cuando `RunTest` devuelve.

Consulte también

[Punteros inteligentes \(C++ moderno\)](#)

Procedimiento Creación y uso de instancias CComPtr y CComQIPtr

06/03/2021 • 5 minutes to read • [Edit Online](#)

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.

```

void CComPtrDemo()
{
    HRESULT hr = CoInitialize(NULL);

    // Declare the smart pointer.
    CComPtr 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 pControl;
    hr = pGraph->QueryInterface(IID_PPV_ARGS(&pControl));
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Obtain a third interface.
    CComPtr 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 parientes 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>`.

```

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();
}

```

Consulta también

[Punteros inteligentes \(C++ moderno\)](#)

Pimpl para encapsulación en tiempo de compilación (C++ moderno)

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 implementación". Es posible que ya esté familiarizado con el concepto, pero lo sepa con otros nombres como Cheshire cat o la expresión de firewall del compilador.

¿Por qué usar pimpl?

A continuación se muestra cómo la expresión pimpl puede mejorar el ciclo de vida de desarrollo de software:

- Minimización de las dependencias de compilación.
- Separación de la interfaz y la implementación.
- Portabilidad.

Encabezado Pimpl

```
// 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 la recompilación en cascada y los diseños de objeto frágiles. Es adecuado para los tipos populares (transitivamente).

Implementación de Pimpl

Defina la `impl` clase en el archivo .cpp.

```
// 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 la posibilidad de agregar compatibilidad para la especialización de intercambio que no inicia.

Consulta también

[Aquí está otra vez C++](#)

Manejo de excepciones en MSVC

16/04/2020 • 3 minutes to read • [Edit Online](#)

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 archivos y las llamadas a funciones realizadas desde otros módulos, son todas las posibles fuentes de excepciones, incluso cuando el programa se está ejecutando correctamente. Un código robusto prevé y controla las excepciones. Para detectar errores lógicos, utilice 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 C++](#)

Para la mayoría de los programas C++, debe usar el control de excepciones C++. Es seguro para tipos y garantiza que los destructores de objetos se invocan durante el desenredado 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 C++ o MFC. SEH solo debe utilizarse en programas que no son de MFC C.

- [Excepciones de MFC](#)

Desde la versión 3.0, MFC ha utilizado excepciones C++. Todavía admite sus macros de control de excepciones más antiguas, que son similares a las excepciones C++ en forma. Para obtener información sobre cómo mezclar macros MFC y excepciones C++, vea [Excepciones: uso de macros MFC y excepciones C++](#).

Utilice 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 estándar de C++ ([/EHsc](#)) es el valor predeterminado en los nuevos proyectos de C++ en Visual Studio.

No se recomienda mezclar los mecanismos de control de excepciones. Por ejemplo, no use excepciones C++ con control de excepciones estructurado. El uso exclusivo del control de excepciones de C++ hace que el código sea más portátil y le permite controlar excepciones de cualquier tipo. Para obtener más información acerca de los inconvenientes del control de excepciones estructurado, vea Control de [excepciones estructurado](#).

En esta sección

- [Prácticas recomendadas modernas de C++ para excepciones y control de errores](#)
- [Cómo diseñar para la seguridad de excepciones](#)
- [Cómo: Interfaz entre código excepcional y no excepcional](#)
- [Las instrucciones try, catch y throw](#)
- [Cómo se evalúan los bloques catch](#)
- [Excepciones y desenredado de pilas](#)

- Especificaciones de excepción
- noexcept
- Excepciones de C++ no controladas
- Mezclar excepciones de C (estructuradas) y de C++
- Control de excepciones estructurado (C/C++)

Consulte también

[Referencia del idioma C++](#)

[Control de excepciones x64](#)

[Manejo de excepciones \(C++/CLI y C++/CX\)](#)

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

02/11/2020 • 14 minutes to read • [Edit Online](#)

En C++ moderno, en la mayoría de los escenarios, la mejor manera de notificar y controlar los errores lógicos y los errores en tiempo de ejecución es usar excepciones. Es especialmente cierto cuando la pila puede contener varias llamadas de 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 el código que detecta errores para pasar la información hacia arriba en la pila de llamadas.

Usar excepciones para código excepcional

Los errores de programa suelen dividirse en dos categorías: errores lógicos causados por errores de programación, por ejemplo, un error "Índice fuera del intervalo". Y, los errores en tiempo de ejecución que están más allá del control del programador, por ejemplo, un error de "servicio de red no disponible". En la programación de estilo C y en COM, el informe de errores se administra 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 llamador puede recuperar opcionalmente después de cada llamada de función para ver si se notificaron errores. Por ejemplo, la programación COM usa el valor devuelto HRESULT para comunicar errores al autor de la llamada. Y la API de Win32 tiene la `GetLastError` función para recuperar el último error que comunicó la pila de llamadas. En ambos casos, el autor de la llamada debe 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 ninguna advertencia. O bien, se puede seguir ejecutando con datos incorrectos y generar resultados incorrectos.

Se prefieren las excepciones en C++ moderno por los siguientes motivos:

- Una excepción obliga al código de llamada 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 permitir que se propague la excepción. No tienen que coordinarse con otras capas.
- El mecanismo de desenredo de pila de excepciones destruye todos los objetos en el ámbito después de que se produzca una excepción, de acuerdo con las reglas bien definidas.
- Una excepción permite una separación limpia entre el código que detecta el error y el código que controla el error.

En el siguiente ejemplo simplificado se muestra la sintaxis necesaria para iniciar y detectar excepciones en 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 `try` bloque, si se *produce* una excepción, se *detectará* por el primer `catch` bloque asociado cuyo tipo coincide con el de la excepción. En otras palabras, la ejecución pasa de la `throw` instrucción a la `catch` instrucción. Si no se encuentra ningún bloque `catch` utilizable, `std::terminate` se invoca y el programa se cierra. En C++, se puede producir cualquier tipo; sin embargo, se recomienda que inicie un tipo que deriva directa o indirectamente de `std::exception`. En el ejemplo anterior, el tipo de excepción, `invalid_argument`, se define en la biblioteca estándar en el `<stdexcept>` archivo de encabezado. C++ no proporciona ni requiere un `finally` bloque para asegurarse de que todos los recursos se liberan si se produce una excepción. La expresión de adquisición de recursos es de inicialización (RAII), que usa punteros inteligentes, proporciona la funcionalidad necesaria para la limpieza de recursos. Para obtener más información, consulte [Cómo: diseñar para la seguridad de excepciones](#). Para obtener información sobre el mecanismo de desenredado de la pila de C++, vea [excepciones y desenredo de pila](#).

Instrucciones básicas

Un control de errores sólido es desafiante en cualquier lenguaje de programación. Aunque las excepciones proporcionan varias características que admiten un buen control de errores, no pueden hacer todo el trabajo por usted. Para obtener las ventajas del mecanismo de excepción, tenga en cuenta las excepciones al diseñar el código.

- Utilice aserciones para comprobar si hay errores que nunca deben producirse. Utilice excepciones para comprobar si hay errores que puedan producirse, por ejemplo, errores en la validación de entrada en parámetros de funciones públicas. Para obtener más información, vea la sección [excepciones frente a aserciones](#).
- Utilice excepciones cuando el código que controla el error se separa del código que detecta el error en una o varias llamadas de función intermedias. Considere la posibilidad de usar códigos de error en lugar de bucles críticos para el rendimiento cuando el código que controla el error está estrechamente acoplado al código que lo detecta.
- Para cada función que puede producir o propagar una excepción, proporcione una de las tres garantías

de excepción: la garantía segura, la garantía básica o la garantía nothrow (noexception). Para obtener más información, consulte [Cómo: diseñar para la seguridad de excepciones](#).

- Producir excepciones por valor, se detectan por referencia. No se detecte lo que no se puede controlar.
- No use especificaciones de excepciones, que están desusadas en C++ 11. Para obtener más información, vea la sección [Especificaciones noexcept de excepciones y](#).
- Use los tipos de excepción de la biblioteca estándar cuando se apliquen. Derivar tipos de excepción personalizados de la jerarquía de [exception clases](#).
- No permita que las excepciones salgan de los destructores o de las 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 cruce seguro y el desenredado de la pila es aproximadamente comparable al costo de una llamada de función. Se requieren estructuras de datos adicionales para realizar el seguimiento de la pila de llamadas después de `try` escribir un bloque, y se requieren instrucciones adicionales para desenredar la pila si se produce una excepción. Sin embargo, en la mayoría de los escenarios, el costo de rendimiento y la superficie de memoria no es significativo. El efecto adverso de las excepciones en el rendimiento es probable que sea significativo solo en sistemas con restricción de memoria. O bien, en los bucles críticos para el rendimiento, en los que es probable que se produzca un error con regularidad y que haya un acoplamiento estrecho 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 ni medir. Incluso en esos raros casos en los que el costo es significativo, puede sopesarlo con la mayor exactitud, mantenimiento más sencillo y otras ventajas que proporciona una directiva de excepciones bien diseñada.

Excepciones frente a aserciones

Las excepciones y aserciones son dos mecanismos distintos para detectar errores en tiempo de ejecución en un programa. Use `assert` instrucciones para comprobar las condiciones durante el desarrollo que nunca deben ser true si todo el código es correcto. No hay ningún punto de controlar este tipo de error mediante el uso de una excepción, porque el error indica que es necesario corregir algo en el código. No representa una condición que el programa tiene que recuperar en tiempo de ejecución. Una `assert` detiene la ejecución en la instrucción para que pueda inspeccionar el estado del programa en el depurador. Una excepción continúa la ejecución del primer controlador Catch adecuado. Utilice 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 solo genera un mensaje en un registro y finaliza el programa. Compruebe siempre los argumentos de las funciones públicas mediante el uso de excepciones. Incluso si la función no tiene errores, es posible que no tenga control total sobre los argumentos que puede pasar un usuario a él.

Excepciones de C++ frente a excepciones 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 son similares a los de las excepciones de C++, salvo que SEH usa las `__try` `__except` construcciones, y `__finally` en lugar de `try` y `catch`. En el compilador de Microsoft C++ (MSVC), las excepciones de C++ se implementan para SEH. Sin embargo, al escribir código de C++, utilice la sintaxis de excepción de C++.

Para obtener más información acerca de SEH, vea [control de excepciones estructurado \(C/C++\)](#).

Especificaciones de excepciones y `noexcept`

Las especificaciones de excepción se introdujeron en C++ como una manera de especificar las excepciones que una función puede iniciar. Sin embargo, las especificaciones de excepción demostraron un problema en la práctica y están en desuso en el borrador estándar de C++ 11. Se recomienda no utilizar `throw` las especificaciones de excepción excepto `throw()`, lo que indica que la función no permite excepciones de escape. Si debe usar especificaciones de excepción del formulario en desuso `throw(type-name)`, el soporte técnico de MSVC es limitado. Para obtener más información, vea [Especificaciones de excepciones \(Throw\)](#). El `noexcept` especificador se introduce en C++ 11 como la alternativa preferida a `throw()`.

Consulte también

[Cómo: interfaz entre código excepcional y no excepcional](#)

[Referencia del lenguaje C++](#)

[Biblioteca estándar de C++](#)

Cómo: diseñar para la seguridad de excepciones

06/03/2021 • 13 minutes to read • [Edit Online](#)

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.

Mantener simples las clases de recursos

Al encapsular la administración manual de recursos en las clases, use una clase que no hace nada excepto administrar un único recurso. Al mantener la clase simple, se reduce el riesgo de introducir pérdidas de recursos. Utilice [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`.

```

// 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) { }
};

```

Usar la expresión RAI para administrar recursos

Para que sea segura para excepciones, una función debe asegurarse de que los objetos que ha asignado mediante `malloc` o `new` se destruyen, y de que todos los recursos como los identificadores de archivo se cierran o se liberan incluso si se produce una excepción. La expresión de *adquisición de recursos* es el modo de inicialización (RAII) vincula la administración de dichos recursos a 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 RAII, 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 RAII tiene una importancia crítica. Tenga en cuenta que las `vector` clases similares de, `string`, `make_shared`, `fstream` y controlan la adquisición del recurso. Sin embargo, `unique_ptr` y `shared_ptr` las construcciones tradicionales son especiales porque la adquisición de recursos la realiza el usuario en lugar del objeto; por lo tanto, cuentan como la *liberación de recursos es la destrucción* pero son cuestionables como RAII.

Las tres garantías de excepción

Normalmente, la seguridad de las excepciones se describe en términos de las tres garantías de excepción que una función puede proporcionar: la *garantía sin errores*, la *garantía segura* y la *garantía básica*.

Garantía sin errores

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 segura

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 la posibilidad de almacenar todo el estado de la clase en un miembro de datos que esté ajustado en un puntero inteligente, especialmente si una clase incluye el concepto de que la "inicialización puede producir errores". Aunque C++ permite miembros de datos sin inicializar, no admite instancias de clase sin inicializar 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.

Consulta también

[Procedimientos recomendados de C++ moderno para excepciones y control de errores](#)

[Cómo: interfaz entre código excepcional y no excepcional](#)

Cómo: interfaz entre código excepcional y no excepcional

06/03/2021 • 10 minutes to read • [Edit Online](#)

En este artículo se describe cómo implementar el control de excepciones coherente en un módulo de C++ y cómo traducir esas excepciones a códigos de error, y viceversa, en los límites de la excepción.

A veces, un módulo de C++ tiene que servir de interfaz con código que no usa las excepciones (código sin excepciones). Este tipo de interfaz se conoce como *límite* de la 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 trivial, probablemente sea preferible tener una directiva coherente de control de errores basada en excepciones. Y probablemente no es conveniente abandonar las excepciones solo porque se interactúa con código sin excepciones; tampoco es conveniente mezclar directivas de error basadas en excepciones y no basadas en excepciones en el módulo de C++.

Llamar 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, decida primero qué tipo de garantías de excepción va a proporcionar: ningún throw, segura 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 utiliza punteros inteligentes o administradores de recursos similares para poseer los recursos. Para obtener más información sobre las consideraciones de diseño, vea [Cómo: diseñar para la seguridad de 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 RAII (Resource Acquisition Is Initialization) para los identificadores de archivo. El constructor detecta una condición de "archivo no encontrado" y produce una excepción para propagar el error en la pila de llamadas del módulo 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 Active Template Library (ATL) `CHandle` para este mismo propósito, o `unique_ptr` junto con un eliminador personalizado). Las funciones que llaman a las API de Win32 y CRT detectan errores y, a continuación, producen excepciones de C++ mediante la función definida localmente `ThrowLastErrorIf`, que a su vez utiliza la `Win32Exception` clase, derivada de la `runtime_error` clase. Todas las funciones de este ejemplo proporcionan una garantía de excepción segura; si se produce una excepción en cualquier momento en estas funciones, no se pierden recursos y no se modifica el estado del programa.

```
// 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.

```

```

        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
    }
}

```

Llamar 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 utilizar 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. Por consiguiente, la función de C++ debe detectar específicamente cada excepción que pueda administrar y, si es necesario, debe convertir la excepción a 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 irre recuperable al llamador, porque el programa podría estar en un estado desconocido.

El ejemplo siguiente muestra una función para la que se supone que cualquier excepción que pueda producirse

es `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.

```
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 convierte de excepciones a códigos de error, un problema potencial se debe a que los códigos de error no contienen a menudo la riqueza de información que una excepción puede almacenar. Para solucionar este error, puede proporcionar un `catch` bloque para cada tipo de excepción específico que se pueda producir y realizar el registro para registrar los detalles de la excepción antes de que se convierta en un código de error. Este enfoque puede crear una gran cantidad de repeticiones de código si varias funciones usan el mismo conjunto de `catch` bloques. Una buena manera de evitar la repetición del código es refactorizar esos bloques en una función de utilidad privada que implementa los `try` `catch` bloques y y acepta un objeto de función que se invoca en el `try` bloque. En cada función pública, pase el código a la función de utilidad como una expresión lambda.

```
template<typename Func>
bool Win32ExceptionBoundary(Func&& f)
{
    try
    {
        return f();
    }
    catch(Win32Exception& e)
    {
        SetLastError(e.GetErrorCode());
    }
    catch(const std::exception& e)
    {
        SetLastError(MY_APPLICATION_GENERAL_ERROR);
    }
    return false;
}
```

En el ejemplo siguiente se muestra cómo escribir la expresión lambda que define el objeto de función (functor). Cuando usa una expresión lambda para definir un functor "alineado", este suele ser más fácil de leer de lo que sería si se escribe como un objeto de función con nombre.

```
bool DiffFiles3(const string& file1, const string& file2)
{
    return Win32ExceptionBoundary([&]() -> bool
    {
        File f1(file1);
        File f2(file2);
        if (IsTextFileDiff(f1, f2))
        {
            SetLastError(MY_APPLICATION_ERROR_FILE_MISMATCH);
            return false;
        }
        return true;
    });
}
```

Para obtener más información sobre las expresiones lambda, vea [expresiones lambda](#).

Consulta también

[Procedimientos recomendados de C++ moderno para excepciones y control de errores](#)
[Procedimientos: Diseño de seguridad de excepciones](#)

Instrucciones try, throw y catch (C++)

06/03/2021 • 5 minutes to read • [Edit Online](#)

Para implementar el control de excepciones en C++, se usan las `try` `throw` expresiones, y `catch`.

En primer lugar, use un `try` bloque para incluir una o varias instrucciones que puedan producir una excepción.

Una `throw` expresión indica que se ha producido una condición excepcional (a menudo, un error) en un `try` bloque. Puede usar un objeto de cualquier tipo como operando de una `throw` expresión. Normalmente, este objeto se emplea para comunicar información sobre el error. En la mayoría de los casos, se recomienda usar la clase [STD:: Exception](#) o una de las clases derivadas definidas en la biblioteca estándar. Si no es adecuado usar una de ellas, se recomienda derivar su propia clase de excepción de `std::exception`.

Para controlar las excepciones que se pueden producir, implemente uno o varios `catch` bloques inmediatamente después de un `try` bloque. Cada `catch` bloque especifica el tipo de excepción que puede controlar.

En este ejemplo se muestra un `try` bloque 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 por `const` referencia en la `catch` instrucción. Se recomienda producir excepciones por valor y detectarlas mediante la referencia `const`.

Ejemplo

```
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");
    // ...
}
```

Observaciones

El código que aparece después de la `try` cláusula es la sección protegida del código. La `throw` expresión *produce*, es decir, produce una excepción. El bloque de código después de la `catch` cláusula es el controlador de excepciones. Este es el controlador que *detecta* la excepción que se produce si los tipos de las `throw` expresiones y `catch` son compatibles. Para obtener una lista de las reglas que rigen la coincidencia de tipos en los `catch` bloques, consulte [cómo se evalúan los bloques Catch](#). Si la `catch` instrucción especifica puntos suspensivos (...) en lugar de un tipo, el `catch` bloque controla cada tipo de excepción. Al compilar con la opción `/EHA`, se pueden incluir excepciones estructuradas de C y excepciones asíncronas generadas por el sistema o generadas por la aplicación, como la protección de memoria, la división por cero y las infracciones de punto flotante. Dado `catch` que los bloques se procesan en el orden del programa para buscar un tipo coincidente, un controlador de puntos suspensivos debe ser el último controlador del `try` bloque asociado. Use `catch(...)` con precaución; no permita que un programa continúe a menos que el bloque `catch` sepa 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 usar este formato al volver a iniciar la excepción, ya que esto conserva la información de tipo polimórfico de la excepción original. Este tipo de expresión solo se debe usar en un `catch` controlador o en una función a la que se llama desde un `catch` controlador. El objeto de excepción que se vuelve a iniciar es el objeto de excepción original, no una copia.

```
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;
}
```

Consulta también

[Procedimientos recomendados de C++ moderno para excepciones y control de errores](#)

[Palabras clave](#)

[Excepciones de C++ no controladas](#)

[__uncaught_exception](#)

Cómo se evalúan los bloques catch (C++)

06/03/2021 • 4 minutes to read • [Edit Online](#)

C++ permite iniciar excepciones de cualquier tipo, aunque en general se recomienda iniciar tipos derivados de `std::exception`. Una excepción de C++ puede ser detectada por un `catch` controlador que especifica el mismo tipo que la excepción iniciada, o por un controlador que puede 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 produce una excepción, puede ser detectada por los siguientes tipos de `catch` Controladores:

- Un controlador que pueda aceptar cualquier tipo (mediante la sintaxis de puntos suspensivos).
- Un controlador que acepta el mismo tipo que el objeto de excepción; Dado que es una copia, `const` `volatile` se omiten los modificadores y.
- Un controlador que acepte una referencia al mismo tipo que el objeto de excepción.
- Un controlador que acepta una referencia a un `const` `volatile` formulario o del mismo tipo que el objeto de excepción.
- Un controlador que acepta una clase base del mismo tipo que el objeto de excepción; Dado que es una copia, `const` `volatile` se omiten los modificadores y. El `catch` controlador de una clase base no debe preceder al `catch` controlador de la clase derivada.
- Un controlador que acepte una referencia a una clase base del mismo tipo que el objeto de excepción.
- Un controlador que acepta una referencia a un `const` `volatile` formulario o 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 `catch` aparecen los controladores es significativo, ya que los controladores de un `try` bloque determinado se examinan en orden de aparición. Por ejemplo, es un error colocar el controlador para una clase base antes del controlador para una clase derivada. Una vez que `catch` se encuentra un controlador coincidente, no se examinan los controladores subsiguientes. Como resultado, un `catch` controlador de puntos suspensivos debe ser el último controlador de su `try` bloque. Por ejemplo:

```
// ...
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 de puntos suspensivos `catch` es el único controlador que se examina.

Consulta también

[Procedimientos recomendados de C++ moderno para excepciones y control de errores](#)

Excepciones y desenredo de pila en C++

06/03/2021 • 7 minutes to read • [Edit Online](#)

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 se encuentran en el ámbito entre las instrucciones Throw y Catch se destruyen en un proceso que se conoce como *desenredado* de la pila. En el desenredo de la pila, la ejecución se desarrolla del modo siguiente:

1. El control alcanza la `try` instrucción mediante la ejecución secuencial normal. Se ejecuta la sección protegida en el `try` bloque.
2. Si no se produce ninguna excepción durante la ejecución de la sección protegida, `catch` no se ejecutan las cláusulas que siguen al `try` bloque. La ejecución continúa en la instrucción después de la última `catch` cláusula que sigue al `try` bloque asociado.
3. Si se produce una excepción durante la ejecución de la sección protegida o en cualquier rutina a la que llame la sección protegida, ya sea directa o indirectamente, se creará un objeto de excepción a partir del objeto creado por el `throw` operando. (Esto implica que un constructor de copias puede estar implicado). En este momento, el compilador busca una `catch` cláusula en un contexto de ejecución superior que pueda controlar una excepción del tipo que se produce, o para un `catch` controlador que pueda controlar cualquier tipo de excepción. Los `catch` controladores se examinan en orden de aparición después del `try` bloque. Si no se encuentra ningún controlador adecuado, se examina el siguiente bloque de inclusión dinámica `try`. Este proceso continúa hasta que se examina el bloque de inclusión más externo `try`.
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 `catch` se encuentra un controlador coincidente y se captura por valor, se inicializa su parámetro formal 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 se han creado por completo, pero que aún no se han desestructurado, entre el principio del `try` bloque que está asociado con el `catch` controlador y el sitio de inicio de la excepción. La destrucción se produce en orden inverso al de construcción. El `catch` controlador se ejecuta y el programa reanuda la ejecución después del último controlador, es decir, en la primera instrucción o construcción que no es un `catch` controlador. El control solo puede escribir un `catch` controlador a través de una excepción producida, nunca a través de una `goto` instrucción o una `case` etiqueta en una `switch` instrucción.

Ejemplo de desenredado de la 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 Diseñar para excepciones. Si marca como comentario la instrucción catch, puede observar lo que sucede cuando un programa finaliza debido a una excepción no controlada.

```
#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 << 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, noexception) (C++)

06/03/2021 • 7 minutes to read • [Edit Online](#)

Las especificaciones de excepción son una característica del lenguaje C++ que indica la intención del programador sobre los tipos de excepción que se pueden propagar por una función. Puede especificar que una función pueda o no salir mediante una excepción mediante una *especificación de excepción*. El compilador puede utilizar esta información para optimizar las llamadas a la función y para finalizar el programa si una excepción inesperada escapa a la función.

Antes de C++ 17 había dos tipos de especificación de excepción. La *especificación noexception* era nueva en c++ 11. Especifica si el conjunto de excepciones potenciales que puede escapar la función está vacío. La *especificación de excepción dinámica*, o `throw(optional_type_list)` especificación, quedó en desuso en c++ 11 y se quitó en c++ 17, a excepción de `throw()`, que es un alias para `noexcept(true)`. Esta especificación de excepción se diseñó para proporcionar información de resumen sobre qué excepciones se pueden iniciar desde una función, pero en la práctica se ha descubierto que es problemática. La especificación de la excepción dinámica que devolvió un poco útil era la especificación incondicional `throw()`. Por ejemplo, la declaración de función:

```
void MyFunction(int i) throw();
```

indica al compilador que la función no produce ninguna excepción. Sin embargo, 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 `noexception` en lugar de uno anterior:

```
void MyFunction(int i) noexcept;
```

En la tabla siguiente se resume la implementación de Microsoft C++ de especificaciones de excepciones:

ESPECIFICACIÓN DE LA EXCEPCIÓN	SIGNIFICADO
<code>noexcept</code> <code>noexcept(true)</code> <code>throw()</code>	La función no produce ninguna excepción. En /STD: el modo c++ 14 (que es el valor predeterminado), <code>noexcept</code> y <code>noexcept(true)</code> son equivalentes. Cuando se produce una excepción de una función que se declara <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 /STD: c++ 14, el resultado es un comportamiento indefinido. No se invoca ninguna función específica. Se trata de una divergencia con respecto al estándar de C++ 14, que requiere que el compilador invoque <code>STD:: inesperado</code> . Visual Studio 2017 versión 15,5 y posteriores: en /STD: el modo c++ 17,, <code>noexcept</code> <code>noexcept(true)</code> y <code>throw()</code> son equivalentes. En /STD: modo c++ 17 , <code>throw()</code> es un alias para <code>noexcept(true)</code> . En el modo /STD: c++ 17 , cuando se produce una excepción desde una función declarada con cualquiera de estas especificaciones, se invoca <code>STD:: Terminate</code> como requiere el estándar c++ 17.

ESPECIFICACIÓN DE LA EXCEPCIÓN	SIGNIFICADO
<pre>noexcept(false) throw(...)</pre> <p>Sin especificación</p>	La función puede producir una excepción de cualquier tipo.
<pre>throw(type)</pre>	(C++ 14 y versiones anteriores) La función puede producir una excepción de tipo <code>type</code> . El compilador acepta la sintaxis, pero lo interpreta como <code>noexcept(false)</code> . En el modo /STD: c++ 17, el compilador emite la advertencia C5040.

Si se usa el control de excepciones en una aplicación, debe haber una función en la pila de llamadas que controle las excepciones iniciadas antes de salir del ámbito externo de una función marcada como `noexcept`, `noexcept(true)` o `throw()`. Si alguna de las funciones llamadas entre la que produce una excepción y la que controla la excepción se especifican como `noexcept`, `noexcept(true)` (o `throw()` en el modo /STD: c++ 17), el programa se termina cuando la función `noexception` propaga la excepción.

El comportamiento de excepción de una función depende de los siguientes factores:

- El [modo de compilación estándar de lenguaje](#) que se establece.
- Si está compilando la función con C o C++.
- Qué opción del compilador [/EH](#) se usa.
- 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 produce excepciones en `/EHsc` y puede producir excepciones estructuradas en `/EHS`, `/EHA` o `/EHAc`.

En la tabla siguiente se resume si una función de C++ puede producirse potencialmente en varias opciones de control de excepciones del compilador:

FUNCIÓN	/EHSC	/EHS	/EHA	/EHAC
Función de C++ sin especificación de excepciones	Sí	Sí	Sí	Sí
Función de C++ con la <code>noexcept</code> , <code>noexcept(true)</code> especificación de excepción, o <code>throw()</code>	No	No	Sí	Sí
Función de C++ con la <code>noexcept(false)</code> , <code>throw(...)</code> especificación de excepción, o <code>throw(type)</code>	Sí	Sí	Sí	Sí

Ejemplo

```

// 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();
}

```

```

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 excepciones y control de errores](#)

noexcept (C++)

06/03/2021 • 3 minutes to read • [Edit Online](#)

C++ 11: Especifica si una función puede producir excepciones.

Sintaxis

noexception-expresión:

`noexcept`

`noexception (Constant-Expression)`

Parámetros

constant-expression

Expresión constante de tipo `bool` que representa si el conjunto de posibles tipos de excepción está vacío. La versión incondicional es equivalente a `noexcept(true)`.

Observaciones

Una expresión `noexception` es un tipo de *especificación de excepción*, un sufijo de una declaración de función que representa un conjunto de tipos que puede coincidir con un controlador de excepciones para cualquier excepción que sale de una función. Operador condicional unario `noexcept(constant_expression)` donde `constant_expression` produce `true`, y su sinónimo incondicional `noexcept`, especifica que el conjunto de posibles tipos de excepción que pueden salir de una función está vacío. Es decir, la función nunca produce una excepción y nunca permite que se propague una excepción fuera de su ámbito. El operador `noexcept(constant_expression)` donde `constant_expression` produce `false` o la ausencia de una especificación de excepción (distinta de para un destructor o una función de desasignación) indica que el conjunto de excepciones potenciales 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, ya sea directa o indirectamente, también son `noexcept` o `const`. El compilador no comprueba necesariamente todas las rutas de acceso de código para las excepciones que podrían propagarse hasta una `noexcept` función. Si una excepción sale del ámbito externo de una función marcada `noexcept`, se invoca a `STD:: Terminate` inmediatamente y no hay ninguna garantía de que se invoquen los destructores de los objetos en el ámbito. Use en `noexcept` lugar del especificador de excepción dinámica `throw()`, que ahora está en desuso en el estándar. Se recomienda aplicar `noexcept` a cualquier función que nunca permita que una excepción se propague hacia arriba en la pila de llamadas. Cuando se declara una función `noexcept`, permite que el compilador genere código más eficaz en varios contextos diferentes. Para obtener más información, vea [Especificaciones de excepciones](#).

Ejemplo

Una función de plantilla que copia su argumento se podría declarar `noexcept` en la condición de que el objeto que se va a copiar sea un tipo de datos anterior (POD) sin formato. Este tipo de función podría declararse de este modo:

```
#include <type_traits>

template <typename T>
T copy_object(const T& obj) noexcept(std::is_pod<T>)
{
    // ...
}
```

Consulta también

[Procedimientos recomendados de C++ moderno para excepciones y control de errores](#)
[Especificaciones de excepciones \(Throw, noexcept\)](#)

Excepciones de C++ no controladas

06/03/2021 • 2 minutes to read • [Edit Online](#)

Si no se encuentra un controlador coincidente (o un `catch` controlador de puntos suspensivos) para la excepción actual, `terminate` se llama a la función predefinida en tiempo de ejecución. (También 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 `terminate` rutina 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`.

```
// 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

```
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`.

Consulta también

[Procedimientos recomendados de C++ moderno para excepciones y control de errores](#)

Mezclar excepciones de C (estructuradas) y de C++

02/11/2020 • 3 minutes to read • [Edit Online](#)

Si desea escribir código portable, no se recomienda el uso del control de excepciones estructurado (SEH) en un programa de C++. Sin embargo, a veces es posible que desee compilar mediante `/EHa` y mezclar excepciones estructuradas y código fuente de C++, y necesitar alguna utilidad para controlar ambos tipos de excepciones. Dado que un controlador de excepciones estructurado no tiene ningún concepto de objetos o excepciones con tipo, no puede controlar las excepciones producidas por el código de C++. Sin embargo, los controladores de C++ `catch` pueden controlar las excepciones estructuradas. `try` `throw` El compilador de C no acepta la sintaxis de control de excepciones de C++ (`,` `catch` `),` `__try` pero `__except` `__finally` el compilador de C++ admite la sintaxis de control de excepciones estructurado (`,`).

Vea [`_set_se_translator`](#) para obtener información sobre cómo controlar las 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 terminación (`__finally` bloques) siempre se ejecutan, incluso durante el desenredado después de que se produzca una excepción.
- El control de excepciones de C++ puede detectar y conservar la semántica de desenredo en todos los módulos compilados con las [`/EH`](#) Opciones del compilador, que habilitan la semántica de desenredo.
- Puede haber algunas situaciones en las que no se llame a las funciones de destructor para todos los objetos. Por ejemplo, podría producirse una excepción estructurada al intentar realizar una llamada de función a través de un puntero de función no inicializado. Si los parámetros de la 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

- [Usar `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 excepciones y control de errores](#)

Usar setjmp y longjmp

06/03/2021 • 2 minutes to read • [Edit Online](#)

Cuando se usan conjuntamente `setjmp` y `longjmp`, proporcionan una forma de ejecutar un no local `goto`. Normalmente se utilizan en código C para pasar el control de ejecución al control de errores o al código de recuperación en una rutina llamada anteriormente sin usar las convenciones de llamada o devolución estándar.

Caution

Dado `setjmp` `longjmp` que y no admiten la destrucción correcta de objetos de marco de pila entre los compiladores de C++, y porque pueden degradar el rendimiento evitando la optimización en variables locales, no se recomienda su uso en programas de C++. En su lugar, se recomienda usar `try` `catch` construcciones y.

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 los objetos locales durante el desenredo de la pila. Sin embargo, si usa `/EHS` o `/EHsc` para compilar, y una de las funciones que usa `noexception` `longjmp`, es posible que no se produzca el desenredado del destructor para esa función, dependiendo del estado del optimizador.

En el código portable, cuando `longjmp` se ejecuta una llamada, la destrucción correcta de objetos basados en fotogramas no está garantizada explícitamente por el estándar y es posible que no sea compatible con otros compiladores. Para informarle, en el nivel de ADVERTENCIA 4, una llamada a `setjmp` produce la advertencia C4611: la interacción entre '`_setjmp`' y la destrucción de objetos de C++ no es portable.

FIN de Específicos de Microsoft

Vea también

[Mezclar excepciones de C \(estructuradas\) y de C++](#)

Controlar excepciones estructuradas en C++

02/11/2020 • 8 minutes to read • [Edit Online](#)

La principal diferencia entre el control de excepciones estructurado de C (SEH) y el control de excepciones de C++ es que el modelo de control de excepciones de C++ trata los tipos, mientras que el modelo de control de excepciones estructurado de C trata las excepciones de un tipo. Concretamente, `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 genera una excepción estructurada en C, cada controlador posible ejecuta un filtro que examina el contexto de la excepción de C y determina si se debe aceptar la excepción, pasársela a otro controlador u omitirla. Cuando se produce una excepción en C++, puede ser de cualquier tipo.

Una segunda diferencia es que el modelo de control de excepciones estructurado de C se conoce como *asincrónico*, porque las excepciones se producen de forma secundaria en el flujo de control normal. El mecanismo de control de excepciones de C++ es totalmente *sincrónico*, lo que significa que las excepciones solo se producen 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 estructurados o `_finally` controladores de terminación estructurados. Para obtener más información, vea [control de excepciones estructurado \(C/C++\)](#).

En la opción del compilador `/EHA`, si se genera una excepción de C en un programa de C++, se puede controlar mediante un controlador de excepciones estructurado con su filtro asociado o un controlador de C++, lo que `catch` se acerque dinámicamente al contexto de la excepción. Por ejemplo, este programa de C++ de ejemplo genera una excepción de C dentro de un contexto de C++ en `try`:

Ejemplo: detectar una excepción de C en un bloque catch de C++

```
// exceptions_Exception_Handling_Differences.cpp
// compile with: /EHa
#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;
    }
}
```

```
In finally.  
Caught a C exception.
```

Clases contenedoras de excepciones de C

En un ejemplo sencillo como el anterior, la excepción de C solo se puede detectar mediante puntos suspensivos (...) `catch` controlador. No se comunica al controlador ninguna información sobre el tipo o la naturaleza de la excepción. Aunque este método funciona, en algunos casos es posible que desee definir una transformación entre los dos modelos de control de excepciones para que cada excepción de C esté asociada a una clase específica. Para transformar uno, puede definir una clase de "contenedor" de excepciones de C, que se puede usar o derivar para atribuir un tipo de clase específico a una excepción de C. Al hacerlo, cada excepción de C se puede controlar por separado mediante un controlador de C++ concreto `catch`, en lugar de todos ellos en 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. También es posible que desee definir un constructor predeterminado y un constructor que acepte un `unsigned int` argumento (para proporcionar la representación subyacente de la excepción de C) y un constructor de copias bit a bit. A continuación se muestra una posible implementación de una clase contenedora de excepciones de 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 usar esta clase, instale una función de traducción de excepciones de C personalizada 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 producir cualquier excepción con tipo (quizás un `SE_Exception` tipo o un tipo de clase derivado de `SE_Exception`) que se puede detectar mediante un controlador de C++ coincidente adecuado `catch`. En su lugar, la función de traducción puede devolver, lo que indica que no se controló la excepción. Si la propia función de traducción genera 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 traducción que escriba se llama una vez para cada invocación de función en la pila que tenga `try` bloques. No hay ninguna función de conversión predeterminada; Si no especifica una llamada a `_set_se_translator`, la excepción de C solo puede ser detectada por un controlador de puntos suspensivos `catch`.

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`:

```

// exceptions_Exception_Handling_Differences3.cpp
// compile with: /EHs
#include <stdio.h>
#include <eh.h>
#include <windows.h>

class SE_Exception {
private:
    SE_Exception() {}
    unsigned int nSE;
public:
    SE_Exception( SE_Exception& e ) : nSE(e.nSE) {}
    SE_Exception(unsigned int n) : nSE(n) {}
    ~SE_Exception() {}
    unsigned int getSeNumber() { return nSE; }
};

void SEFunc() {
    __try {
        int x, y = 0;
        x = 5 / y;
    }
    __finally {
        printf_s( "In finally\n" );
    }
}

void trans_func( unsigned int u, _EXCEPTION_POINTERS* pExp ) {
    printf_s( "In trans_func.\n" );
    throw SE_Exception( u );
}

int main() {
    _set_se_translator( trans_func );
    try {
        SEFunc();
    }
    catch( SE_Exception e ) {
        printf_s( "Caught a __try exception with SE_Exception.\n" );
        printf_s( "nSE = 0x%x\n", e.getSeNumber() );
    }
}

```

```

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++)

02/11/2020 • 8 minutes to read • [Edit Online](#)

El control de excepciones estructurado (SEH) es una extensión de Microsoft a C que controla ciertas situaciones de código excepcionales, como errores de hardware, sin problemas. Aunque Windows y Microsoft C++ admiten SEH, se recomienda usar el control de excepciones de C++ estándar de ISO. Hace que el código sea más portátil y flexible. Sin embargo, para mantener el código existente o para determinados tipos de programas, es posible que tenga que usar SEH.

Específico de Microsoft:

Gramática

```
try-except-statement :  
    __try compound-statement __except ( expression ) compound-statement  
  
try-finally-statement :  
    __try compound-statement __finally compound-statement
```

Observaciones

Con SEH, puede asegurarse de que los recursos, como los bloques de memoria y los archivos, se liberan correctamente si la ejecución finaliza inesperadamente. También puede controlar problemas específicos (por ejemplo, memoria insuficiente) mediante el uso de código estructurado conciso que no depende de `goto` instrucciones o de probar los códigos de retorno.

Las `try-except` instrucciones y a las `try-finally` que se hace referencia en este artículo son extensiones de Microsoft para el lenguaje 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 utiliza SEH en un programa de C++ que se compila mediante la opción `/EHs` o `/EHsc`, se llama a los destructores para los objetos locales pero otros comportamientos de ejecución podrían no ser los esperados. Para ver una ilustración, vea 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 de ISO, que también admite el compilador de Microsoft C++. 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 de C que usa SEH, puede combinarlo con código de C++ que usa el control de excepciones de C++. Para obtener más información, vea [controlar excepciones estructuradas en C++](#).

Existen dos mecanismos de SEH:

- [Controladores de excepciones](#), o `__except` bloques, que pueden responder o descartar la excepción.
- [Controladores de terminación](#), o `__finally` bloques, a los que se llama siempre, independientemente de que una excepción cause la terminación o no.

Estos dos tipos de controladores son distintos, pero están estrechamente relacionados a través de un proceso conocido como *desenredado 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 reconocer la excepción y pasar el control a otros controladores.
- Reconocer la excepción pero descartarla.
- Reconocer la excepción y controlarla.

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 mucho más alta en la pila. La función que se está ejecutando actualmente y todas las demás funciones del marco de pila finalizan. Durante este proceso, se *desenreda* la pila. 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 el uso de un controlador de terminación, limpia los recursos que de otro modo permanecerán abiertos debido a una terminación anómala. Si ha escrito una sección crítica, puede salir en el controlador de terminación. Cuando el programa se va a cerrar, puede 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 indicó anteriormente, se llama a los destructores para los objetos locales si se utiliza SEH en un programa de C++ y se compila mediante la `/EHs` `/EHsc` opción o. Sin embargo, el comportamiento durante la ejecución puede no ser el esperado si también se usan excepciones de C++. En este ejemplo se muestran estas diferencias de comportamiento.

```

#include <stdio.h>
#include <Windows.h>
#include <exception>

class TestClass
{
public:
    ~TestClass()
    {
        printf("Destroying TestClass!\r\n");
    }
};

__declspec(noinline) void TestCPPEX()
{
#ifdef CPPEX
    printf("Throwing C++ exception\r\n");
    throw std::exception("");
#else
    printf("Triggering SEH exception\r\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\r\n");
    }

    return 0;
}

```

Si utiliza `/EHsc` para compilar este código pero la macro de control de pruebas local `CPPEX` no está definida, el `TestClass` destructor no se ejecuta. El resultado tendrá un aspecto similar al siguiente:

```

Triggering SEH exception
Executing SEH __except block

```

Si utiliza `/EHsc` para compilar el código y `CPPEX` se define mediante `/DCPPEX` (para que se produzca una excepción de C++), el `TestClass` destructor se ejecuta y el resultado tiene el siguiente aspecto:

```

Throwing C++ exception
Destroying TestClass!
Executing SEH __except block

```

Si utiliza `/EHa` para compilar el código, el `TestClass` destructor ejecuta si la excepción se produjo mediante `std::throw` o mediante SEH para desencadenar la excepción. Es decir, si `CPPEX` está definido o no. El resultado tendrá un aspecto similar al siguiente:

```
Throwing C++ exception
Destroying TestClass!
Executing SEH __except block
```

Para obtener más información, vea [/EH](#) (modelo de control de excepciones).

FIN de específico de Microsoft

Consulte también

[Control de excepciones](#)

[Palabras clave](#)

[`<exception>`](#)

[Errores y control de excepciones](#)

[Control de excepciones estructurado \(Windows\)](#)

Escribir un controlador de excepciones

06/03/2021 • 2 minutes to read • [Edit Online](#)

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](#)

Consulta también

[Structured Exception Handling \(C/C++\)](#)

Instrucción `try-except`

02/11/2020 • 9 minutes to read • [Edit Online](#)

La `try-except` instrucción es una extensión **específica de Microsoft** que admite el control de excepciones estructurado en los lenguajes C y C++.

```
// . . .
__try {
    // guarded code
}
__except ( /* filter expression */ ) {
    // termination code
}
// . . .
```

Gramática

```
try-except-statement :
    __try compound-statement | __except ( expression ) compound-statement
```

Observaciones

La `try-except` instrucción es una extensión de Microsoft para los lenguajes C y C++. Permite a las aplicaciones de destino obtener el control cuando se producen eventos que normalmente finalizan la ejecución del programa. Estos eventos se denominan *excepciones estructuradas* o *excepciones* para abreviar. El mecanismo que se encarga de estas excepciones se denomina *control de excepciones estructurado* (SEH).

Para obtener información relacionada, vea la [instrucción try-finally](#).

Las excepciones pueden estar basadas en hardware o en software. El control estructurado de excepciones es útil incluso cuando las aplicaciones no pueden recuperarse 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.

NOTE

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. En el caso de los programas de C++, se recomienda usar las instrucciones de control de excepciones de C++ nativo: [try](#), [Catch](#) y [Throw](#).

La instrucción compuesta después de la `__try` cláusula es el *cuerpo* o la sección *protegida*. La `__except` expresión también se conoce como 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 deben llevarse a cabo si se produce una excepción durante la ejecución de la sección del cuerpo. 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 a la que llame la sección protegida, `__except` se evalúa la expresión. Hay tres valores posibles:

- `EXCEPTION_CONTINUE_EXECUTION` (-1) Se descartará la excepción. La ejecución continúa en el punto donde se ha producido la excepción.
- `EXCEPTION_CONTINUE_SEARCH` (0) no se reconoce la excepción. Siga buscando en la pila un controlador, primero para las instrucciones contenedoras `try-except` y, a continuación, para los controladores con la siguiente prioridad más alta.
- `EXCEPTION_EXECUTE_HANDLER` (1) se reconoce la excepción. Transfiera el control al controlador de excepciones ejecutando la `__except` instrucción compuesta y después continúe la ejecución después del `__except` bloque.

La `__except` expresión se evalúa como una expresión de C. Está limitado a un único 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 a una `try` instrucción, pero es válido para saltar de una. No se llama al controlador de excepciones si un proceso finaliza en medio de ejecutar una `try-except` instrucción.

Por compatibilidad con versiones anteriores, `_try`, `_excepty_leave` son sinónimos para `__try`, `__except` y `__leave` a menos que se especifique la opción del compilador [/za \(deshabilitar extensiones de lenguaje\)](#).

La `__leave` palabra clave

La `__leave` palabra clave solo es válida dentro de la sección protegida de una `try-except` instrucción y su efecto es saltar al final de la sección protegida. La ejecución de la primera instrucción continúa después del controlador de excepciones.

Una `goto` instrucción también puede saltar fuera de la sección protegida y no degrada el rendimiento como en una instrucción `try-finally`. Esto se debe a que no se produce el desenredado de la pila. Sin embargo, se recomienda usar la `__leave` palabra clave en lugar de una `goto` instrucción. La razón es que es menos probable que se produzca un error de programación si la sección protegida es grande o compleja.

Funciones intrínsecas de control de excepciones estructurado

El control de excepciones estructurado proporciona dos funciones intrínsecas que están disponibles para su uso con la `try-except` instrucción: [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 de `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:

```
typedef struct _EXCEPTION_POINTERS {
    PEXCEPTION_RECORD ExceptionRecord;
    PCONTEXT ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

Los tipos `PEXCEPTION_RECORD` de puntero 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` dentro del controlador de excepciones. Sin embargo, solo puede usar `GetExceptionInformation` dentro de la expresión de filtro de excepciones. La información a la que apunta está generalmente 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

```
// 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");
}
```

Salida

```
hello
in try
in try
in filter.
caught AV as expected.
in finally. termination:
    abnormal
in except
world
```

Vea también

[Escribir un controlador de excepciones](#)

[Control de excepciones estructurado \(C/C++\)](#)

[Palabras clave](#)

Escribir un filtro de excepción

06/03/2021 • 4 minutes to read • [Edit Online](#)

Para controlar una excepción puede saltar al nivel del controlador de la excepción o puede continuar la ejecución. En lugar de usar el código del controlador de excepciones para controlar la excepción y pasar a través de, puede usar una expresión de *filtro* para limpiar el problema. A continuación, devolviendo `EXCEPTION_CONTINUE_EXECUTION` (-1), puede reanudar el flujo normal sin borrar la pila.

NOTE

Algunas excepciones impiden que continúe el flujo. Si el *filtro* se evalúa como -1 para este tipo de excepción, el sistema genera una nueva excepción. Cuando llame a `RaiseException`, determine si la excepción continuará.

Por ejemplo, el código siguiente usa una llamada de función en la expresión de *filtro*: esta función controla el problema y, a continuación, devuelve -1 para reanudar el flujo de control normal:

```
// 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 de función en la expresión de *filtro* siempre que el *filtro* necesite hacer algo complejo. La evaluación de la expresión hace que se ejecute la función (en este caso, `Eval_Exception`).

Tenga en cuenta el uso de `GetExceptionCode` para determinar la excepción. Se debe llamar a esta función dentro de la expresión de filtro de la `__except` instrucción. `Eval_Exception` no se puede llamar a `GetExceptionCode`, pero debe tener el código de excepción que se le ha pasado.

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 `__except` bloque de instrucciones, que en este ejemplo está vacío, nunca se puede ejecutar 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. Se puede usar para comprobar un código de retorno específico y, a continuación, devolver uno de dos valores distintos. Por ejemplo, el filtro del código siguiente reconoce la excepción solo si la excepción es `STATUS_INTEGER_OVERFLOW` :

```
__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:

```
__except( GetExceptionCode() == STATUS_INTEGER_OVERFLOW ) {
```

El operador condicional es más útil en situaciones en las que se desea que el filtro se evalúe como-1 `EXCEPTION_CONTINUE_EXECUTION` .

El operador de coma permite ejecutar varias expresiones en la secuencia. A continuación, devuelve el valor de la última expresión. Por ejemplo, el código siguiente almacena el código de excepción en una variable y después lo comprueba:

```
__except( nCode = GetExceptionCode(), nCode == STATUS_INTEGER_OVERFLOW )
```

Consulte también

[Escribir un controlador de excepciones](#)
[Structured Exception Handling \(C/C++\)](#)

Generar excepciones de software

06/03/2021 • 4 minutes to read • [Edit Online](#)

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 una excepción si detecta esa condición en el código y, a continuación, la notifica 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` al detectar un problema.
- Use filtros de control de excepciones para probar el código de excepción definido.

El `<winerror.h>` archivo 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 tanto, el código de error resultante debe tener los cuatro bits más altos establecidos en hexadecimal E. Por ejemplo, las siguientes definiciones 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).

```
#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 genera la `STATUS_INSUFFICIENT_MEM` excepción en respuesta a un problema de asignación de memoria:

```
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 el Windows SDK para obtener más información.

En los filtros de control de excepciones, puede probar los códigos que haya definido. Por ejemplo:

```
__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

06/03/2021 • 3 minutes to read • [Edit Online](#)

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 los procesadores Intel 80 x86).
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.
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, puede tomar las precauciones adecuadas contra el posible efecto de estos errores de bajo nivel escribiendo [controladores de terminación](#).

Consulta también

[Escribir un controlador de excepciones](#)

[Structured Exception Handling \(C/C++\)](#)

Restricciones de los controladores de excepciones

02/11/2020 • 2 minutes to read • [Edit Online](#)

La limitación principal al uso de controladores de excepciones en el código es que no se puede utilizar una `goto` instrucción para saltar a un `try` bloque de instrucciones. En su lugar, se debe especificar el bloque de instrucciones a través del flujo de control normal. Puede saltar fuera de un `try` bloque de instrucciones y puede anidar los controladores de excepciones que elija.

Consulte también

[Escribir un controlador de excepciones](#)

[Control de excepciones estructurado \(C/C++\)](#)

Escribir un controlador de finalización

06/03/2021 • 2 minutes to read • [Edit Online](#)

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](#)
- [Limpieza de recursos](#)
- [Cronología de las acciones en el control de excepciones](#)
- [Restricciones de los controladores de finalización](#)

Consulta también

[Structured Exception Handling \(C/C++\)](#)

Instrucción `try-finally`

02/11/2020 • 8 minutes to read • [Edit Online](#)

La `try-finally` instrucción es una extensión **específica de Microsoft** que admite el control de excepciones estructurado en los lenguajes C y C++.

Sintaxis

La sintaxis siguiente describe la `try-finally` instrucción:

```
// . . .
__try {
    // guarded code
}
__finally {
    // termination code
}
// . . .
```

Gramática

```
try-finally-statement :
    __try compound-statement __finally compound-statement
```

La `try-finally` instrucción es una extensión de Microsoft para los lenguajes C y C++ que permiten a las aplicaciones de destino garantizar 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, vea [try-except Statement](#). Para obtener más información sobre el control de excepciones estructurado en general, vea [control de excepciones estructurado](#). Para obtener más información sobre el control de excepciones en aplicaciones administradas con C++/CLI, vea [control de excepciones en /clr](#).

NOTE

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. En el caso de los programas de C++, se recomienda utilizar el mecanismo de control de excepciones de C++ (instrucciones `try`, `catch` y `throw`).

La instrucción compuesta después de la `__try` cláusula 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, independientemente de que salga de la sección protegida mediante una excepción (terminación anómala) o mediante el paso estándar (terminación normal).

El control llega a una `__try` instrucción mediante la ejecución secuencial simple (paso a través). Cuando el

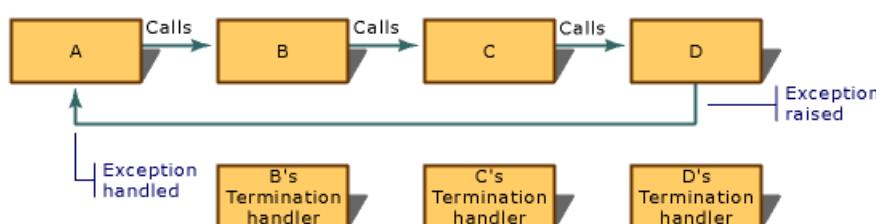
control entra `__try` en, su controlador asociado se activa. Si el flujo de control alcanza el final del bloque `try`, la ejecución continúa del modo siguiente:

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 finaliza (por ejemplo, a través `goto` de un fuera del cuerpo protegido o una `return` instrucción), el controlador de terminación se ejecuta *antes* de que el flujo de control salga de la sección protegida.

Una `__finally` instrucción no bloquea la búsqueda de un controlador de excepciones adecuado.

Si se produce una excepción en el `__try` bloque, el sistema operativo debe encontrar un controlador para la excepción o se producirá un error en el programa. Si se encuentra un controlador, se ejecutan todos y cada uno `__finally` de los bloques y se reanuda la ejecución 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

NOTE

El comportamiento de `try-finally` es diferente de otros lenguajes que admiten el uso de `finally`, como C#. Un único `__try` puede tener cualquiera de los valores, pero no ambos, de `__finally` y `__except`. 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.

Por compatibilidad con versiones anteriores, `__try`, `__finally` y `__leave` son sinónimos para `__try`, `__finally` y `__leave` a menos que se especifique la opción del compilador [/za](#) (deshabilitar extensiones de lenguaje).

La palabra clave `__leave`

La `__leave` palabra clave solo es válida dentro de la sección protegida de una `try-finally` instrucción y su efecto es saltar al final de la sección protegida. La ejecución continúa en la primera instrucción del controlador de finalización.

Una `goto` instrucción también puede saltar fuera de la sección protegida, pero degrada el rendimiento porque invoca el desenredo de la pila. La `__leave` instrucción es más eficaz porque no produce el desenredado de la pila.

Finalización anómala

Salir de una `try-finally` instrucción mediante la función en tiempo de ejecución `longjmp` se considera una terminación anómala. No es válido saltar a una `__try` instrucción, pero es legal salir de una `__finally`. Se deben ejecutar todas las instrucciones activas entre el punto de salida (terminación normal del `__try` bloque) y

el destino (el `__except` bloque que controla la excepción). Se denomina *desenredado local*.

Si un `__try` bloque finaliza prematuramente por cualquier motivo, incluido un salto fuera del bloque, el sistema ejecuta el `__finally` bloque asociado como parte del proceso de desenredado de la pila. En tales casos, la `AbnormalTermination` función devuelve `true` si se llama desde dentro del `__finally` bloque; de lo contrario, devuelve `false`.

No se llama al controlador de terminación si un proceso se elimina en medio de la ejecución de una `try-finally` instrucción.

FIN de específico de Microsoft

Consulte también

[Escribir un controlador de finalización](#)

[Control de excepciones estructurado \(C/C++\)](#)

[Palabras clave](#)

[Sintaxis del controlador de terminación](#)

Limpiar recursos

02/11/2020 • 2 minutes to read • [Edit Online](#)

Durante la ejecución del controlador de terminación, es posible que no sepa qué recursos se han adquirido antes de llamar al controlador de terminación. Es posible que el `__try` bloque de instrucciones se interrumpiera antes de adquirir todos los recursos, de modo que no se abriera ningún recurso.

Para ser seguro, debe comprobar qué recursos están abiertos antes de continuar con la limpieza de control de terminación. Un procedimiento recomendado es:

1. Inicializar los identificadores en NULL.
2. En el `__try` bloque de instrucciones, adquiera recursos. Los identificadores se establecen en valores positivos a medida que se adquiere el recurso.
3. En el `__finally` bloque de instrucciones, libere cada recurso cuyo identificador o variable de marca correspondiente sea distinto de cero o no sea NULL.

Ejemplo

Por ejemplo, el código siguiente usa un controlador de terminación para cerrar tres archivos y liberar un bloque de memoria. Estos recursos se adquirieron en el `__try` bloque de instrucciones. Antes de limpiar un recurso, el código comprueba primero para ver si se ha adquirido el recurso.

```
// 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", "wt" );
    }
    __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](#)

[Control de excepciones estructurado \(C/C++\)](#)

Resumen sobre los intervalos de control de excepciones Resumen

02/11/2020 • 3 minutes to read • [Edit Online](#)

Un controlador de terminación se ejecuta independientemente de cómo `__try` finalice el bloque de instrucciones. Las causas incluyen saltar fuera del `__try` bloque, una `longjmp` instrucción que transfiere el control fuera del bloque y desenredar la pila debido al control de excepciones.

NOTE

El compilador de Microsoft C++ admite dos formas de las `setjmp` `longjmp` instrucciones y. 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 examina la jerarquía de controladores de excepciones activos y ejecuta el filtro del controlador con mayor prioridad. Este es el controlador de excepciones instalado más recientemente y más profundamente anidado, que va por los bloques y las llamadas a funciones.
3. Si este filtro pasa el control (devuelve 0), el proceso continúa hasta que se encuentra un filtro que no pasa el control.
4. Si este filtro devuelve -1, la ejecución continúa donde se produjo la excepción y no se realiza ninguna terminación.
5. Si el filtro devuelve 1, se producen los eventos siguientes:
 - El sistema desenreda la pila: borra todos los marcos de pila entre los que 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](#)
[Control de excepciones estructurado \(C/C++\)](#)

Restricciones de los controladores de finalización

02/11/2020 • 2 minutes to read • [Edit Online](#)

No se puede usar una `goto` instrucción para saltar a un bloque de `__try` instrucciones o un `__finally` bloque de instrucciones. En su lugar, se debe especificar el bloque de instrucciones a través del flujo de control normal. (Sin embargo, puede saltar fuera de un `__try` bloque de instrucciones). Además, no puede anidar un controlador de excepciones ni un controlador de finalización dentro de un `__finally` bloque.

Algunos tipos de código permitidos en un controlador de terminación producen resultados cuestionables, por lo que debe usarlos con precaución, si es que lo hace. Una es una `goto` instrucción que salta fuera de un `__finally` bloque de instrucciones. Si el bloque se ejecuta como parte de la finalización normal, no sucede nada inusual. Pero si el sistema está desenredando la pila, se detiene el desenredado. A continuación, la función actual gana el control como si no hubiese una terminación anómala.

Una `return` instrucción dentro de un `__finally` bloque de instrucciones presenta aproximadamente la misma situación. El control vuelve al llamador inmediato de la función que contiene el controlador de finalización. Si el sistema estaba Desenredando la pila, este proceso se detiene. Después, el programa continúa como si no se hubiera producido ninguna excepción.

Consulte también

[Escribir un controlador de finalización](#)

[Control de excepciones estructurado \(C/C++\)](#)

Transportar excepciones entre subprocessos

06/03/2021 • 20 minutes to read • [Edit Online](#)

El compilador de Microsoft C++ (MSVC) admite el *transporte de 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 excepciones de transporte, MSVC proporciona el tipo `exception_ptr` y las funciones `current_exception`, `rethrow_exception` y `make_exception_ptr`.

Sintaxis

```
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

no especificado

Clase interna sin especificar que se utiliza para implementar el tipo `exception_ptr`.

m

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 `make_exception_ptr` función devuelve un `exception_ptr` objeto que hace referencia a la excepción especificada por el parámetro *e*.

Observaciones

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 `catch` bloque y, a continuación, llamar a la `current_exception` función para almacenar la excepción en un `exception_ptr` objeto. 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 *transporte una excepción* significa que una excepción en 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.

Finalmente, el subprocesso principal puede detectar la excepción actual en un `catch` bloque y, a continuación, procesarla o iniciarla 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 `catch` instrucción pueden transportar excepciones de SEH y C++.
- Las opciones del compilador `/EHA`, `/EHS` y `/EHsc` y la `catch` instrucción pueden transportar excepciones de C++.
- La opción del compilador `/CLR` y la `catch` instrucción pueden transportar excepciones de C++. La opción del compilador `/CLR` implica la especificación de la opción `/EHA`. 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 desplazar entre subprocessos mediante las funciones de Common Language Runtime.

IMPORTANT

Se recomienda especificar la opción del compilador /EHsc y detectar solo las excepciones de C++. Se expone a una amenaza de seguridad si se usa la opción del compilador /EHA o /CLR y una `catch` instrucción con una *declaración de excepción ()* de puntos suspensivos `catch(...)`. Probablemente pretenda usar la `catch` instrucción para capturar algunas excepciones específicas. 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 secciones siguientes se describe cómo transportar excepciones utilizando el `exception_ptr` tipo y las `current_exception` funciones, `rethrow_exception` y `make_exception_ptr`.

tipo de 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 `current_exception` `make_exception_ptr` función o 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 los operadores de acceso a miembros de puntero (`->`) o de direccionamiento indirecto (`*`). 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.

current_exception función)

Llame a la `current_exception` función en un `catch` bloque. Si una excepción está en vuelo y el `catch` bloque puede detectar la excepción, la `current_exception` función devuelve un `exception_ptr` objeto que hace referencia a la excepción. De lo contrario, la función devuelve un objeto `exception_ptr` NULL.

Detalles

La `current_exception` función captura la excepción que está en vuelo independientemente de si la `catch`

instrucción especifica una instrucción [de declaración de excepción](#).

Se llama al destructor de la excepción actual al final del `catch` bloque si no se 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 usa la opción del compilador `/EHA`, puede detectar una excepción SEH en un `catch` bloque de C++. La función `current_exception` devuelve un objeto `exception_ptr` que hace referencia a la excepción SEH. Y la `rethrow_exception` función produce la excepción SEH si se llama con `exception_ptr` el objeto de transporte como su argumento.

La `current_exception` función devuelve un valor null `exception_ptr` si se llama en un `_finally` controlador de finalización SEH, un controlador de `_except` excepciones o la expresión de `_except` filtro.

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 acerca de las funciones de traductor de SE, consulte [_set_se_translator](#).

rethrow_exception función)

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 `rethrow_exception` función es un valor null `exception_ptr`, 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 detecta la excepción, puede controlarla inmediatamente o usar una `throw` instrucción 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 `make_exception_ptr` función es equivalente a iniciar una excepción de C++, detectarla en un `catch` bloque y, a continuación, llamar a la `current_exception` función para devolver un `exception_ptr` objeto 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.

```
// 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;
}

```

```

exception_ptr 0: Caught an invalid_argument exception.
exception_ptr 1: Caught a myException exception.

```

Requisitos

Encabezado:<exception>

Consulta también

[Control de excepciones](#)

[/EH \(modelo de control de excepciones\)](#)

[/clr \(Compilación de Common Language Runtime\)](#)

Aserción y mensajes proporcionados por el usuario (C++)

06/03/2021 • 3 minutes to read • [Edit Online](#)

El lenguaje C++ admite tres mecanismos de control de errores que ayudan a depurar la aplicación: la [Directiva de #error](#), la palabra clave [Static_assert](#) y la [macro Assert, _assert _wassert](#) macro. Los tres mecanismos emiten mensajes de error y dos de ellos también prueban las aseraciones 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.

Observaciones

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 en vigor en el tiempo de 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 en vigor en tiempo de 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 `static_assert` declaración es especialmente útil para depurar plantillas, porque los argumentos de plantilla se pueden incluir en la expresión especificada por el usuario.

- La [macro Assert, _assert, _wassert](#) macro está en vigor en tiempo de 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 [_ASERTE](#), se asemejan a esta macro pero emiten diferentes mensajes de diagnóstico definidos por el sistema o definidos por el usuario.

Consulta también

[#error \(Directiva\) \(C/C++\)](#)
[Macro Assert, _assert, _wassert](#)
[_ASSERT, _ASERTE, _ASSERT_EXPR macros](#)
[static_assert](#)
[_STATIC_ASSERT \(Macro\)](#)
[Templates \(Plantillas \[C++\]\)](#)

static_assert

06/03/2021 • 6 minutes to read • [Edit Online](#)

Comprueba una aserción de software en tiempo de compilación. Si la expresión constante especificada es `false`, el compilador muestra el mensaje especificado, si se proporciona uno, y se produce el 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), se muestra el parámetro *String-literal* y se produce un error en la compilación. Si la expresión es distinto de cero (true), la `static_assert` declaración 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 [caracteres anchos o multibyte](#).

Observaciones

El parámetro *Constant-Expression* de una `static_assert` declaración representa una *aserción de software*. Una aserción 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 `static_assert` declaración no tiene ningún efecto. Si la condición es falsa, se produce un error en la aserción, el compilador muestra el mensaje en el parámetro de *literal de cadena* y se produce un error en la compilación. En Visual Studio 2017 y versiones posteriores, el parámetro String-literal es opcional.

La `static_assert` declaración comprueba una aserción de software en tiempo de compilación. En cambio, la [macro Assert y las funciones _assert y _wassert](#) prueban una aserción de software en tiempo de ejecución y incurren en tiempo de ejecución en el espacio o en el tiempo. La `static_assert` declaración es especialmente útil para depurar plantillas, porque los argumentos de plantilla se pueden incluir en el parámetro *Constant-Expression*.

El compilador examina la `static_assert` declaración de errores de sintaxis cuando se encuentra la declaración. El compilador evalúa el parámetro *Constant-Expression* inmediatamente si 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 `static_assert` palabra clave en el ámbito de espacio de nombres, clase o bloque. (La `static_assert` palabra clave 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 `static_assert` declaración 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

```
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 `static_assert` declaración tiene ámbito de clase. `static_assert` Comprueba que un parámetro de plantilla es un tipo de *datos antiguos sin formato* (POD). El compilador examina la `static_assert` declaración cuando se declara, pero no evalúa el parámetro *Constant-Expression* hasta que `basic_string` se crea una instancia de la plantilla de clase en `main()`.

Ejemplo: `static_assert` con ámbito de clase

```
#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 `static_assert` declaración tiene ámbito de bloque. `static_assert` Comprueba que el tamaño de la estructura `vmpage` sea igual a la paginación de la memoria virtual del sistema.

Ejemplo: `static_assert` en el ámbito de bloque

```
#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.");
        // ...
    }
    // ...
};
```

Consulta también

[Aserción y mensajes proporcionados por el usuario \(C++\)](#)

[#error \(Directiva\) \(C/C++\)](#)

[Macro Assert,_assert,_wassert](#)

[Templates \(Plantillas \[C++\]\)](#)

[Juego de caracteres ASCII](#)

[Declaraciones y definiciones](#)

Información general de los módulos en C++

02/11/2020 • 14 minutes to read • [Edit Online](#)

C++ 20 presenta *módulos*, una solución moderna para la componentización de bibliotecas y programas de C++. Un módulo es un conjunto de archivos de código fuente que se compilan de forma independiente de las [unidades de traducción](#) que los importan. Los módulos eliminan o reducen considerablemente muchos de los problemas asociados con el uso de archivos de encabezado y también pueden reducir los tiempos de compilación. Las macros, las directivas de preprocesador y los nombres no exportados declarados en un módulo no son visibles y, por lo tanto, no tienen ningún 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 macros. Las declaraciones de la unidad de traducción de importación no participan en la resolución de sobrecarga o en la búsqueda de nombres en el módulo importado. Después de compilar un módulo una vez, los resultados se almacenan en un archivo binario que describe todos los tipos, funciones y plantillas exportados. Ese archivo se puede procesar mucho más rápido que un archivo de encabezado y el compilador puede reutilizarlo cada lugar en el que se importe el módulo en un proyecto.

Los módulos se pueden usar en paralelo con los archivos de encabezado. Un archivo de código fuente de C++ puede importar módulos y también #include archivos de encabezado. En algunos casos, el preprocesador puede importar un archivo de encabezado como un módulo en lugar de #included textualmente. Se recomienda que los nuevos proyectos usen módulos en lugar de archivos de encabezado lo máximo posible. En el caso de los proyectos existentes de mayor tamaño en desarrollo activo, se recomienda experimentar con la conversión de encabezados heredados a módulos para ver si obtiene una reducción significativa en los tiempos de compilación.

Habilitar módulos en el compilador de Microsoft C++

A partir de la versión 16,2 de Visual Studio 2019, los módulos no se implementan totalmente en el compilador de Microsoft C++. Puede usar la característica módulos para crear módulos de una sola partición e importar los módulos de la biblioteca estándar que proporciona Microsoft. Para habilitar la compatibilidad con los módulos, compile con [/experimental: module](#) y [/STD: c + + latest](#). En un proyecto de Visual Studio, haga clic con el botón secundario en el nodo del proyecto en **Explorador de soluciones** y elija **propiedades**. Establezca la lista desplegable **configuración** en **todas las configuraciones**, a continuación, elija **propiedades de configuración > lenguaje C/c++ > Language > Habilitar módulos c++ (experimental)**.

Un módulo y el código que lo consume deben compilarse con las mismas opciones del compilador.

Usar la biblioteca estándar de C++ como módulos

Aunque no se especifica en el estándar C++ 20, Microsoft permite que la implementación de la biblioteca estándar de C++ se importe como módulos. Al importar la biblioteca estándar de C++ como módulos en lugar de #including a través de los archivos de encabezado, puede acelerar los tiempos de compilación según el tamaño del proyecto. La biblioteca se pone en componente en los siguientes módulos:

- 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, basta con agregar una declaración de importación en la parte superior del archivo de código fuente. Por ejemplo:

```
import std.core;
import std.regex;
```

Para consumir el módulo 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 **foo. IXX**. La extensión **. IXX** es necesaria para los archivos de interfaz de módulo en Visual Studio. En este ejemplo, el archivo de interfaz contiene la definición de función, así como la declaración. Sin embargo, las definiciones también se pueden colocar en uno o varios archivos independientes (como se muestra en un ejemplo posterior). La instrucción **Export Module foo** indica que este archivo es la interfaz principal para un módulo denominado **Foo**. El **export** modificador en **f()** indica que esta función estará visible cuando **Foo** lo importe otro programa o módulo. Tenga en cuenta que el módulo hace referencia a un espacio de nombres **Bar**.

```
export module Foo;

#define ANSWER 42

namespace Bar
{
    int f_internal() {
        return ANSWER;
    }

    export int f() {
        return f_internal();
    }
}
```

El archivo **Program. cpp** usa la declaración de **importación** para obtener acceso al nombre que exporta **Foo**. Tenga en cuenta que el nombre **Bar** es visible aquí, pero no todos sus miembros. Tenga en cuenta también que la macro **ANSWER** no está visible.

```
import Foo;
import std.core;

using namespace std;

int main()
{
    cout << "The result of f() is " << Bar::f() << endl; // 42
    // int i = Bar::f_internal(); // C2039
    // int j = ANSWER; //C2065
}
```

La declaración de importación solo puede aparecer en el ámbito global.

Implementar módulos

Puede crear un módulo con un solo archivo de interfaz (. IXX) que exporta nombres e incluye implementaciones de todos los tipos y funciones. También puede colocar las implementaciones en uno o varios archivos de implementación independientes, de forma similar a como se usan los archivos. h y. cpp. La `export` palabra clave solo se usa en el archivo de interfaz. Un archivo de implementación puede **importar** otro módulo, pero no los `export` nombres. Los archivos de implementación se pueden denominar con cualquier extensión. Un archivo de interfaz y el conjunto de archivos de implementación que respaldan se tratan como una clase especial de unidad de traducción denominada *unidad de módulo*. Un nombre que se declara en cualquier archivo de implementación es visible automáticamente en todos los demás archivos de la misma unidad de módulo.

En el caso de los módulos de mayor tamaño, puede dividir el módulo en varias unidades de módulo denominadas *particiones*. Cada partición consta de un archivo de interfaz respaldado por uno o varios archivos de implementación. (A partir de la versión 16,2 de Visual Studio 2019, las particiones aún no están totalmente implementadas).

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 un espacio de nombres se exporta explícitamente, se exportan todas las declaraciones dentro de esa definición de espacio de nombres.

Al realizar una búsqueda dependiente de argumentos para las resoluciones de sobrecarga en la unidad de traducción de importación, el compilador tiene en cuenta las funciones que se declaran en la misma unidad de traducción (incluidas las interfaces de módulo) en que se define el tipo de los argumentos de la función.

Particiones de módulos

NOTE

Esta sección se proporciona por integridad. Las particiones aún no se han implementado en el compilador de Microsoft C++.

Un módulo se puede dividir en *particiones*, cada uno de los cuales consta de un archivo de interfaz y cero o más archivos de implementación. Una partición de módulo es similar a un módulo, salvo que comparte la propiedad de todas las declaraciones en todo el módulo. Todos los nombres exportados por los archivos de la interfaz de partición se importan y se vuelven a exportar mediante el archivo de interfaz principal. El nombre de una partición debe comenzar con el nombre del módulo seguido de un signo de dos puntos. Las declaraciones en cualquiera de las particiones están visibles en todo el módulo. No se necesitan precauciones especiales para evitar errores de la regla de una definición (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:

```
module Foo:part1
```

y el archivo de interfaz de partición se inicia de la siguiente manera:

```
export module Foo:part1
```

Para obtener acceso a las declaraciones de otra partición, una partición debe importarla, pero solo puede usar el nombre de la partición, no el nombre del módulo:

```
module Foo: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:

```
export import :part1
export import :part2
...
```

La unidad de interfaz principal puede importar archivos de implementación de particiones, pero no puede exportarlos porque no se permite que los nombres exporten ningún nombre. Esto permite a un módulo mantener los detalles de implementación internos en el módulo.

Módulos y archivos de encabezado

Puede incluir los archivos de encabezado en un archivo de origen de módulo colocando la `#include` Directiva antes de la declaración de módulo. Estos archivos se consideran en el fragmento de *módulo global*. Un módulo solo puede ver los nombres en el *fragmento de módulo global* que se encuentran en los encabezados que incluye explícitamente. El fragmento de módulo global solo contiene los símbolos que se usan realmente.

```
// MyModuleA.cpp

#include "customlib.h"
#include "anotherlib.h"

import std.core;
import MyModuleB;

//... rest of file
```

Puede usar un archivo de encabezado tradicional para controlar los módulos que se importan:

```
// MyProgram.h
import std.core;
#ifndef DEBUG_LOGGING
import std.filesystem;
#endif
```

Archivos de encabezado importados

NOTE

Esta sección solo es meramente informativa. Las importaciones heredadas todavía no se han implementado en el compilador de Microsoft C++.

Algunos encabezados son suficientemente independientes para que se les permita usar la palabra clave `Import`. La diferencia principal entre un encabezado importado y un módulo importado es que las definiciones de preprocesador del encabezado están visibles en el programa de importación inmediatamente después de la instrucción `Import`. (Las definiciones de preprocesador de los archivos incluidos en ese encabezado *no* son visibles).

```
import <vector>
import "myheader.h"
```

Consulte también

[módulo, importar, exportar](#)

módulo, importar, exportar

16/04/2021 • 2 minutes to read • [Edit Online](#)

Las **declaraciones** del módulo, **importar** y están disponibles en C++20 y requieren el modificador del compilador `export /experimental:module` junto con `/std:c++latest`. Para obtener más información, vea [Introducción a los módulos en C++](#).

module

Coloque una **declaración de módulo** al principio de un archivo de implementación de módulo para especificar que el contenido del archivo pertenece al módulo con nombre.

```
module ModuleA;
```

exportar

Use una **declaración de módulo** de exportación para el archivo de interfaz principal del módulo, que debe tener la extensión .ixx:

```
export module ModuleA;
```

En un archivo de interfaz, use el modificador en los nombres que están `export` diseñados para formar parte de la interfaz pública:

```
// ModuleA.ixx

export module ModuleA;

namespace Bar
{
    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:

```
//MyProgram.cpp

import module ModuleA;

int main() {
    Bar::f(); // OK
    Bar::d(); // OK
    Bar::internal_f(); // Ill-formed: error C2065: 'internal_f': undeclared identifier
}
```

Es `export` posible que la palabra clave 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.

importación

Use una **declaración de importación** para que los nombres de un módulo sean visibles en el programa. La declaración de importación debe aparecer después de la declaración del módulo y después de #include directivas, pero antes de cualquier declaración del archivo.

```
module ModuleA;

#include "custom-lib.h"
import std.core;
import std.regex;
import ModuleB;

// begin declarations here:
template <class T>
class Baz
{...};
```

Observaciones

Tanto la **importación** como el **módulo** se tratan como palabras clave solo cuando aparecen al principio de una línea lógica:

```
// 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++, la **importación** de tokens y el módulo siempre son identificadores y nunca palabras clave cuando se usan como argumentos para una macro.

Ejemplo

```
#define foo(...) __VA_ARGS__
foo(
import // Always an identifier, never a keyword
)
```

Fin específico de Microsoft

Consulte también

[Información general de los módulos en C++](#)

Plantillas (C++)

06/03/2021 • 13 minutes to read • [Edit Online](#)

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, declarado explícitamente por el programador o deducidos por el compilador. Sin embargo, muchas estructuras de datos y algoritmos tienen el mismo aspecto, independientemente del tipo en el que estén trabajando. Las plantillas le permiten definir las operaciones de una clase o función y permitir que el usuario especifique en qué tipos concretos deben funcionar esas operaciones.

Definir y usar plantillas

Una plantilla es una construcción que genera un tipo o una función ordinarios en tiempo de compilación basándose en los argumentos que el usuario proporciona para los parámetros de plantilla. Por ejemplo, puede definir una plantilla de función similar a la siguiente:

```
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 solo parámetro de tipo *T*, cuyo valor devuelto y los parámetros de llamada (LHS y RHS) son todo este tipo. Puede asignar el nombre que deseé a un parámetro de tipo, pero por Convención se usa con más frecuencia las mayúsculas y minúsculas. *T* es un parámetro de plantilla; la `typename` palabra clave indica que este parámetro es un marcador de posición para un tipo. Cuando se llama a la función, el compilador reemplazará cada instancia de `T` con 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 denomina *creación de instancias de plantilla*. `minimum<int>` es una instancia de la plantilla `minimum<T>`.

En cualquier otro lugar, un usuario puede declarar una instancia de la plantilla especializada en `int`. Suponga que `get_a()` y `get_b()` son funciones que devuelven un valor `int`:

```
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 de `T` los argumentos *a* y *b*, puede llamarlo como una función normal:

```
int i = minimum(a, b);
```

Cuando el compilador encuentra la última instrucción, genera una nueva función en la que todas las apariciones de *T* de la plantilla se reemplazan por `int`:

```
int minimum(const int& lhs, const int& rhs)
{
    return lhs < rhs ? lhs : rhs;
}
```

Las reglas para el modo en que el compilador realiza la deducción de tipos en las plantillas de función se basan en las reglas de las funciones ordinarias. Para obtener más información, consulte la [resolución de sobrecarga de llamadas de plantilla de función](#).

Parámetros de tipo

En la `minimum` plantilla anterior, tenga en cuenta que el parámetro de tipo `T` no se califica de ningún modo hasta que se use en los parámetros de llamada de función, donde se agregan los calificadores `const` y `Reference`.

No hay ningún límite práctico para el número de parámetros de tipo. Separe varios parámetros por comas:

```
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:

```
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:

```
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, puede usar `STD:: Vector` en la biblioteca estándar para almacenar variables de tipo `int`, `double`, `STD:: String`, `MyClass`, `const MyClass *`, `MyClass&`, etc. La restricción principal cuando se usan 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` con `MyClass` como en este ejemplo:

```
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 es necesario que los argumentos de tipo de ninguna plantilla determinada pertenezcan a la misma jerarquía de objetos, aunque se puede definir una plantilla que aplique este tipo de restricción. Puede combinar técnicas

orientadas a objetos con plantillas; por ejemplo, puede almacenar un * derivado en un vector <Base*> . Tenga en cuenta que los argumentos deben ser punteros

```
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 `T` se pueden asignar y copiar en copias.

Parámetros sin tipo

A diferencia de los tipos genéricos en otros lenguajes como C# y Java, las plantillas de C++ admiten *parámetros sin tipo*, también denominados parámetros de valor. Por ejemplo, puede proporcionar un valor entero constante para especificar la longitud de una matriz, como en este ejemplo, que es similar a la clase [STD:: Array](#) de la biblioteca estándar:

```
template<typename T, size_t L>
class MyArray
{
    T arr[L];
public:
    MyArray() { ... }
};
```

Observe la sintaxis de la declaración de plantilla. El `size_t` valor se pasa como un argumento de plantilla en tiempo de compilación y debe `const` ser o una `constexpr` expresión. Se utiliza de la siguiente manera:

```
MyArray<MyClass*, 10> arr;
```

Otros tipos de valores, incluidos los punteros y las referencias, se pueden pasar como parámetros que no son de tipo. Por ejemplo, puede pasar un puntero a una función o a un objeto de función para personalizar alguna operación dentro del código de la plantilla.

Deducción de tipos para parámetros de plantilla sin tipo

En Visual Studio 2017 y versiones posteriores, en el modo `/STD: c++ 17`, el compilador deduce el tipo de un argumento de plantilla sin tipo que se declara con `auto`:

```
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`:

```

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, no se necesitan sus nombres de parámetro. De hecho, es un error hacer referencia a los nombres de los parámetros de clase o TypeName de *ARR* desde dentro del cuerpo de `MyClass2`. Por esta razón, se pueden omitir los nombres de parámetro de tipo de *ARR*, tal como se muestra en este ejemplo:

```

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, puede dejarla sin especificar cuando la utiliza. Por ejemplo, la plantilla STD:: Vector tiene un argumento predeterminado para el asignador:

```
template <class T, class Allocator = allocator<T>> class vector;
```

En la mayoría de los casos, la clase predeterminada STD:: allocator es aceptable, por lo que se usa un vector similar al siguiente:

```
vector<int> myInts;
```

Pero si es necesario, puede especificar un asignador personalizado como el siguiente:

```
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 configurados de forma predeterminada, use corchetes angulares vacíos:

```

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 o deseable que una plantilla defina exactamente el mismo código para cualquier tipo. Por ejemplo, puede que desee definir una ruta de acceso de código para que se ejecute 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, puede definir una *especialización* de la plantilla para ese tipo en particular. 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 se especializan todos los parámetros son *especializaciones completas*. Si solo algunos de los parámetros están especializados, se denomina *especialización parcial*.

```
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 se deben declarar en el mismo espacio de nombres que la plantilla original.

Para obtener más información, vea [especialización de plantilla](#).

typename

06/03/2021 • 2 minutes to read • [Edit Online](#)

En las definiciones de plantilla, 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;
```

Observaciones

Esta 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 es dependiente. Para obtener más información, vea [plantillas y resolución de nombres](#).

`typename` cualquier tipo puede usarse en cualquier parte de una declaración o definición de plantilla. No se permite en la lista de clases base, salvo como argumento de plantilla de una clase base de plantilla.

```
template <class T>
class C1 : typename T::InnerType // Error - typename not allowed.
{};
template <class T>
class C2 : A<typename T::InnerType> // typename OK.
{};


```

La `typename` palabra clave también se puede usar en lugar de `class` en las listas de parámetros de plantilla. Por ejemplo, las siguientes instrucciones son semánticamente equivalentes:

```
template<class T1, class T2>...
template<typename T1, typename T2>...
```

Ejemplo

```
// 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

06/03/2021 • 9 minutes to read • [Edit Online](#)

En este tema se describen las reglas que son 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.

```
// 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()
{
}
```

Observe que, 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, especificando parámetros adicionales, como en el ejemplo siguiente.

```

// 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 explican 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.

```

// 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();
}

```

```
// 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()

```

No se permite que las clases locales tengan plantillas de miembro.

Plantillas Friend

Las plantillas de clase pueden tener [amigos](#). 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.

```
// 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. En este caso, se debe colocar la definición de especialización de plantilla friend fuera de la clase de plantilla.

```

// 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.

```

// 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:

```
// 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()
{
}
```

Consulta también

[Templates](#) (Plantillas [C++])

Plantillas de función

06/03/2021 • 3 minutes to read • [Edit Online](#)

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:

```
// 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. A partir de esta plantilla, puede generar funciones que intercambiarán `int` `long` tipos y y también tipos definidos por el usuario. `MySwap` intercambiará incluso las clases si el constructor de copias y el operador de asignación de la clase se definen correctamente.

Además, la plantilla de función impedirá que intercambie objetos de tipos diferentes, porque 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:

```
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:

```
// 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 se convertirá `j` al tipo `char`.

Consulta también

[Templates \(Plantillas \[C++\]\)](#)

[Creación de instancias de plantillas de función](#)

[Creación de instancias explícita](#)

[Especialización explícita de plantillas de función](#)

Crear instancias de plantillas de función

06/03/2021 • 2 minutes to read • [Edit Online](#)

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:

```
// 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()
{
}
```

Consulta también

[Plantillas de función](#)

creación de instancias explícita

06/03/2021 • 2 minutes to read • [Edit Online](#)

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. Como esto es útil cuando se crean archivos de biblioteca (.lib) que utilizan plantillas para la distribución, las definiciones de plantillas sin instancias no se colocan en archivos objeto (.obj).

Este código crea instancias explícitamente `MyStack` para `int` las variables y seis elementos:

```
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:

```
template MyStack<int, 6>::MyStack( void );
```

Puede crear explícitamente instancias de las plantillas de función mediante un argumento de tipo específico para volver a declararlas, tal como se muestra en el ejemplo de [creación de instancias de plantillas de función](#).

Puede usar la `extern` palabra clave para evitar la creación automática de instancias de miembros. Por ejemplo:

```
extern template class MyStack<int, 6>;
```

Del mismo modo, puede marcar determinados miembros como externos y no crear instancias de ellos:

```
extern template MyStack<int, 6>::MyStack( void );
```

Puede usar la `extern` palabra clave para evitar que el compilador genere el mismo código de creación de instancias en más de un módulo de objeto. Para crear instancias de la función de plantilla se deben utilizar los parámetros de plantilla explícitos especificados en al menos un módulo vinculado si se llama a la función; de lo contrario, se producirá un error del vinculador cuando se compile el programa.

NOTE

La `extern` palabra clave de la especialización solo se aplica a las funciones 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.

Consulta también

[Plantillas de función](#)

Especialización explícita de las plantillas de función

06/03/2021 • 2 minutes to read • [Edit Online](#)

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:

```
template<> void MySwap(double a, double b);
```

Esta declaración permite definir una función diferente para `double` las variables. Al igual que las funciones que no son de plantilla, se aplican las conversiones de tipo estándar (como la promoción de una variable de tipo `float` a `double`).

Ejemplo

```
// 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++)

06/03/2021 • 4 minutes to read • [Edit Online](#)

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 función de plantilla más especializada que esté disponible entre las posibles coincidencias. Por ejemplo, si una plantilla de función toma un tipo `T` y otra plantilla de función que toma `T*` está disponible, `T*` se dice que la versión es más especializada. Es preferible usar la `T` versión genérica siempre que el argumento sea un tipo de puntero, aunque ambos serían coincidencias permitidas.

Utilice el proceso siguiente para determinar si un candidato de plantilla de función es más especializado:

1. Considere dos plantillas de función, T1 y T2.
2. Reemplace los parámetros de T1 con un tipo único hipotético X.
3. Con la lista de parámetros de T1, vea si T2 es una plantilla válida para esa lista de parámetros. Omítala cualquier conversión implícita.
4. Repita el mismo proceso con T1 y T2 a la inversa.
5. Si una plantilla es una lista de argumentos de plantilla válida para la otra plantilla, pero el opuesto no es cierto, esa plantilla se considera menos especializada que la otra plantilla. Si usa el paso anterior, ambas plantillas forman argumentos válidos entre sí, se considera que son igualmente especializadas y una llamada ambigua se produce cuando intenta usarlas.
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 sola `T`, porque un tipo hipotético `x*` 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 como se especifica en el estándar:

```
// 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

```
Less specialized function called
More specialized function called
Even more specialized function for const T*
```

Vea también

[Plantillas de función](#)

Plantillas de función miembro

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 funciones de plantilla que son miembros de una clase o una plantilla de clase.

Las funciones miembro pueden ser plantillas de función en varios contextos. Todas las funciones de plantillas de clase son genéricas pero no se hace referencia a ellas como plantillas de miembro o plantillas de función miembro. Si estas funciones miembro tienen sus propios argumentos de plantilla, se consideran plantillas de función miembro.

Ejemplo: declarar plantillas de función miembro

Las plantillas de función miembro de clases de plantilla o clases que no son de plantilla se declaran como plantillas de función con sus parámetros de plantilla.

```
// 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 clase de plantilla

En el ejemplo siguiente se muestra una plantilla de función miembro de una clase de plantilla.

```
// member_function_templates2.cpp
template<typename T>
class X
{
public:
    template<typename U>
    void mf(const U &u)
    {
    }
};

int main()
```

Ejemplo: definir plantillas de miembro fuera de la clase

```
// 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 definida por el usuario con plantilla

No se permite que las clases locales tengan plantillas de miembro.

Las funciones de plantilla de miembro no pueden ser funciones virtuales y no pueden invalidar 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:

```
// 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>.
}
```

Consulta también

[Plantillas de función](#)

Especialización de plantilla (C++)

06/03/2021 • 7 minutes to read • [Edit Online](#)

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

```
// partial_specialization_of_class_templates.cpp
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{};

extern "C" int printf_s(const char*,...);

int main() {
    printf_s("PTS<S>::IsPointer == %d PTS<S>::IsPointerToDataMember == %d\n",
            PTS<S>::IsPointer, PTS<S>:: IsPointerToDataMember);
    printf_s("PTS<S*>::IsPointer == %d PTS<S*>::IsPointerToDataMember ==%d\n"
            , PTS<S*>::IsPointer, PTS<S*>:: IsPointerToDataMember);
    printf_s("PTS<int S::*>::IsPointer == %d PTS"
            "<int S::*>::IsPointerToDataMember == %d\n",
            PTS<int S::*>::IsPointer, PTS<int S::*>::
            IsPointerToDataMember);
}
```

```
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 tome 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`.

```
// 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);
    p = NULL;
    xp.add(p);
    xp.print();
}
}

```

```

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 cualesquiera y después define una especialización parcial de esa clase de plantilla especializada para 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.

```

// 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<char*, char*>* dict = new Dictionary<char*, char*>(10);
    dict->print();
    dict->add("apple", "fruit");
    dict->add("banana", "fruit");
    dict->add("dog", "animal");
    dict->print();

    Dictionary<int, char*>* dict_specialized = new Dictionary<int, char*>(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();
}

```

```

{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

06/03/2021 • 4 minutes to read • [Edit Online](#)

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:

```
T
```

- Un nombre completo con una clasificación que incluye un tipo dependiente:

```
T::myType
```

- Un nombre completo si la parte incompleta identifica un tipo dependiente:

```
N::T
```

- Un tipo const o volatile para el que el tipo base es un tipo dependiente:

```
const T
```

- Un tipo de puntero, referencia, matriz o puntero de función basado en un tipo dependiente:

```
T *, T &, T [10], T (*)()
```

- Una matriz cuyo tamaño se basa en un parámetro de plantilla:

```
template <int arg> class X {  
    int x[arg] ; // dependent type  
}
```

- Un tipo de plantilla creado a partir de un parámetro de plantilla:

```
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.

Consulta también

[Templates](#) (Plantillas [C++])

Resolución de nombres para los tipos dependientes

06/03/2021 • 3 minutes to read • [Edit Online](#)

Use `typename` para nombres completos en las definiciones de plantilla para indicar al compilador que el nombre completo especificado identifica un tipo. Para obtener más información, consulte [TypeName](#).

```
// 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.");
}
```

Name resolved by using typename keyword.

La búsqueda de nombre para los nombres dependientes examina los nombres desde el contexto de la definición de plantilla, en el ejemplo siguiente, este contexto encontraría, `myFunction(char)` y el contexto de la creación de instancias de la plantilla. En el ejemplo siguiente, se crea una instancia de la plantilla en Main; por lo tanto, `MyNamespace::myFunction` es visible desde el punto de creación de instancias y se selecciona 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.

```

// 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

```
Int MyNamespace::myFunction
```

Desambiguación de plantilla

Visual Studio 2012 aplica las reglas estándar de C++ 98/03/11 para la anulación de ambigüedades con la palabra clave "template". En el ejemplo siguiente, Visual Studio 2010 aceptaría las líneas no compatibles y las líneas de ajuste. Visual Studio 2012 solo acepta las líneas de ajuste.

```

#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.

Consulta también

[Resolución de nombres](#)

Resolución de nombres declarados localmente

06/03/2021 • 5 minutes to read • [Edit Online](#)

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 especialización parcial

En el código siguiente se muestra que el nombre de una plantilla de clase A se interpreta de manera diferente en el ámbito de una especialización o especialización parcial.

```
// 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

En el caso de un conflicto de nombres entre un parámetro de plantilla y otro objeto, el parámetro de plantilla se 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.

```

// 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: definir una función miembro fuera de la plantilla de clase

Al definir las funciones miembro de una plantilla fuera de la plantilla de clase, se puede usar un nombre de parámetro de plantilla diferente. Si la definición de la función miembro de la plantilla utiliza un nombre diferente para el parámetro de plantilla que la declaración, y el nombre utilizado en la definición está en conflicto con otro miembro de la declaración, el miembro de la declaración de plantilla tiene prioridad.

```

// 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();
}

```

Z::Z()

Ejemplo: definición de una función de plantilla o miembro fuera del espacio de nombres

Al definir una función de plantilla o una función miembro fuera del espacio de nombres en el que se declaró la plantilla, el argumento de plantilla tiene prioridad sobre los nombres de otros miembros del espacio de nombres.

```

// 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();
}

```

C<T>::g

Ejemplo: la clase base o el nombre de miembro ocultan 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 tiene el mismo nombre que un argumento de plantilla, el nombre de la clase base o miembro oculta el argumento de plantilla.

```

// 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;
}

```

Base

1

Consulta también

[Resolución de nombres](#)

Resolución de sobrecarga de llamadas de plantilla de función

06/03/2021 • 2 minutes to read • [Edit Online](#)

Una plantilla de función puede sobrecargar funciones que no son de plantilla con el mismo nombre. En este escenario, las llamadas a función se resuelven utilizando primero la deducción de argumento de plantilla para crear instancias de la plantilla de función con una especialización exclusiva. Si la deducción de argumento de plantilla no se produce correctamente, se considera que las demás sobrecargas de función resuelven la llamada. Estas otras sobrecargas, también conocidas como conjunto de candidato, incluyen funciones que no son de plantilla y otras plantillas de función de las que se pueden crear instancias. Si la deducción de argumento de plantilla se realiza correctamente, la función generada se compara con las demás funciones para determinar la mejor coincidencia, de acuerdo con las reglas para la resolución de sobrecarga. Para obtener más información, vea [sobrecarga de funciones](#).

Ejemplo: elegir una función que no es de plantilla

Si una función que no es de plantilla constituye una coincidencia igualmente buena con una función de plantilla, se elige la primera (a menos que los argumentos de plantilla estén especificados explícitamente), como en la llamada a `f(1, 1)` en el ejemplo siguiente.

```
// 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 nontemplate function.
    f('a', 1); // Chooses the template function.
    f<int, int>(2, 2); // Template arguments explicitly specified.
}
```

```
f(int, int)
void f(T1, T2)
void f(T1, T2)
```

Ejemplo: la función de plantilla de coincidencia exacta preferida

En el ejemplo siguiente se muestra que se prefiere la función de plantilla que coincide exactamente si la función que no es de plantilla requiere una conversión.

```
// 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 template function f(long, int) because f(int, int)
    // would require a conversion from long to int.
    f(l, i);
}
```

```
void f(T1, T2)
```

Consulta también

[Resolución de nombres](#)

[nombreDeTipo](#)

Organización de código fuente (plantillas de C++)

06/03/2021 • 7 minutes to read • [Edit Online](#)

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 de *creación de instancias explícita*. En el modelo de inclusión, se incluyen las definiciones de miembro en cada archivo 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 impacto 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.

Información previa

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, esto no es necesariamente un error porque las funciones pueden definirse en otra unidad de traducción, en cuyo caso el vinculador las encontrará. Si el vinculador 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. Este es el enfoque usado en la biblioteca estándar.

```

#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 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 `.cpp` archivo que `#includes` el encabezado obtendrá su propia copia de las plantillas de función y de todas las definiciones. El vinculador 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 definitivamente el conjunto de tipos que se usarán para crear una instancia de una plantilla, puede separar el código de plantilla en un `.h` `.cpp` archivo y, en el archivo, `.cpp` crear explícitamente instancias de las plantillas. Esto hará que se genere 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. Esto puede ser un tipo o miembro. Si crea una instancia de un tipo explícitamente, se crean instancias de todos los miembros.

```
template MyArray<double, 5>;
```

```

//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

//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. Un `MyArray` solo se puede usar para `double` los `String` tipos o.

NOTE

En C++ 11, la `export` palabra clave quedó 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

06/03/2021 • 2 minutes to read • [Edit Online](#)

El control de eventos se admite principalmente para las clases COM (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](#).

También se admite el control de eventos para las clases de C++ nativo (clases de C++ que no implementan objetos COM). La compatibilidad con el control de eventos de C++ nativo está en desuso y se quitará en una versión futura. Para obtener más información, vea [control de eventos en C++ nativo](#).

NOTE

Los atributos de evento en C++ nativo son incompatibles con C++ estándar. No se compilan cuando se especifica el `/permissive-` modo de cumplimiento.

El control de eventos admite el uso de un solo multiproceso. Protege los datos del acceso multiproceso simultáneo. Puede derivar las subclases de las clases de origen o receptor de eventos. Estas subclases admiten el suministro de eventos extendidos y la recepción.

El compilador de 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
<code>event_source</code>	Crea un origen de eventos.
<code>event_receiver</code>	Crea un receptor de eventos (receptor).
<code>_event</code>	Declara un evento.
<code>_raise</code>	Resalta el sitio de llamada de un evento.
<code>_hook</code>	Asocia un método de control a un evento.
<code>_unhook</code>	Desasocia un método de controlador de un evento.

Vea también

[Referencia del lenguaje C++](#)

[Palabras clave](#)

`_event` palabra clave

06/03/2021 • 6 minutes to read • [Edit Online](#)

Declara un evento.

NOTE

Los atributos de evento en C++ nativo son incompatibles con C++ estándar. No se compilan cuando se especifica el `/permissive-` modo de cumplimiento.

Sintaxis

```
_event member-function-declarator ;
_event _interface interface-specifier ;
_event data-member-declarator ;
```

Observaciones

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 puede usar la `_event` palabra clave 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 `_event` palabra clave, se llama a todos los controladores de eventos enlazados a ese evento después de que se llame al evento.

Una `_event` declaración de función miembro no puede tener una definición; se genera implícitamente una definición, por lo que se puede llamar a la función miembro de evento como si fuera cualquier función de miembro normal.

NOTE

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 otros delegados.

```
// Examples of native C++ events:  
__event void OnDoubleClick();  
__event HRESULT OnClick(int* b, char* s);
```

Vea [control de eventos en C++ nativo](#) para obtener código de ejemplo.

Eventos COM

Los eventos COM son interfaces. Los parámetros de una función miembro en una interfaz de origen de eventos deben estar *en* parámetros, pero no se aplican rigurosamente. Se debe a que un parámetro *out* no es útil cuando se *realiza* la multidifusión. Se emite una advertencia de nivel 1 si se usa un parámetro de *salida*.

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, se trata de una condición de error. El evento que se provoca anula las llamadas a los otros delegados. El compilador marca automáticamente una interfaz de origen [source](#) de eventos como en el IDL generado.

La `__interface` palabra clave siempre es necesaria después `__event` de para un origen de eventos com.

```
// Example of a COM event:  
__event __interface IEvent1;
```

Vea [control de eventos en com](#) para ver el código de ejemplo.

Eventos administrados

Para obtener información sobre cómo codificar eventos en la nueva sintaxis, vea [Event](#).

Los eventos administrados son miembros de datos o funciones miembro. Cuando se usa con un evento, el tipo de valor devuelto de un delegado debe ser compatible con el [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, vea [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 ello, Predefina el tipo de delegado para declarar un miembro de datos de evento administrado. En cambio, 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:

```
// 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 los `add` `remove` descriptores de acceso y 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

```

// EventHandling_Native_Event.cpp
// compile with: /c
[event_source(native)]
class CSource {
public:
    __event void MyEvent(int nValue);
};

```

Ejemplo: eventos COM

```

// EventHandling_COM_Event.cpp
// compile with: /c
#define _ATL_ATTRIBUTES 1
#include <atbase.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;
    }
};

```

Vea también

[Palabra](#)

[Control de eventos](#)

[event_source](#)
[event_receiver](#)
[__hook](#)
[__unhook](#)
[__raise](#)



palabra clave

06/03/2021 • 5 minutes to read • [Edit Online](#)

Asocia un método de control a un evento.

NOTE

Los atributos de evento en C++ nativo son incompatibles con C++ estándar. No se compilan cuando se especifica el `/permissive-` modo de cumplimiento.

Sintaxis

```
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 de eventos y `EventMethod` es el evento.
- Eventos COM: `SourceClass` es la interfaz de origen de eventos y `EventMethod` es uno de sus métodos.
- Eventos administrados: `SourceClass` es la clase de origen de eventos y `EventMethod` es el evento.

`interface`

El nombre de la interfaz a la que se enlaza `receiver`, solo para los receptores de eventos com en los que el `Layout_dependent` parámetro del `event_receiver` atributo 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.
- `IUnknown` Puntero basado en (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 se especifica el nombre de clase, `__hook` se supone que la clase es aquella desde la que se llama.

- Eventos de C++ nativo: `ReceiverClass` es la clase de 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 de receptor de eventos y `HandlerMethod` es el controlador.

`receiver`

Opta Un puntero a una instancia de la clase de receptor de eventos. Si no se especifica un receptor, el valor predeterminado es la clase o estructura del receptor en la que `_hook` se llama a.

Uso

Se puede usar en cualquier ámbito de función, incluida main, fuera de la clase del receptor de eventos.

Observaciones

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 (cuatro argumentos) en la mayoría de los casos, en concreto, para los receptores de eventos COM en los que el parámetro `layout_dependent` del `event_receiver` atributo es `false`.

En estos casos, no es necesario enlazar todos los métodos en una interfaz antes de activar un evento en uno de los métodos. Solo tiene que enlazar el método que controla el evento. Puede usar el segundo formulario (dos argumentos) de `_hook` solo para un receptor de eventos com en el que `Layout_dependent = true`.

`_hook` Devuelve un valor 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 coincide con la firma del delegado.

Puede llamar a `_hook` y `_unhook` fuera del receptor de eventos, excepto para los eventos com.

Una alternativa al uso de `_hook` es usar el operador `+ =`.

Para obtener información sobre la codificación de eventos administrados en la nueva sintaxis, vea [event](#).

NOTE

Una clase o struct basada en plantilla no puede contener eventos.

Ejemplo

Vea [control de eventos en C++ nativo](#) y [control de eventos en com](#) para obtener ejemplos.

Vea también

[Palabra](#)

[Control de eventos](#)

event_source

event_receiver

__event

__unhook

__raise

`__raise` palabra clave

06/03/2021 • 2 minutes to read • [Edit Online](#)

Resalta el sitio de llamada de un evento.

NOTE

Los atributos de evento en C++ nativo son incompatibles con C++ estándar. No se compilan cuando se especifica el `/permissive-` modo de cumplimiento.

Sintaxis

```
__raise [method-declarator] ;
```

Observaciones

En el 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 emita un error si se llama a un evento que no es de.

NOTE

Una clase o struct basada en plantilla no puede contener eventos.

Ejemplo

```
// 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
}
```

Vea también

Palabra

Control de eventos

`__event`

`__hook`

`__unhook`

Extensiones de componentes de .NET y UWP

`__unhook` palabra clave

06/03/2021 • 5 minutes to read • [Edit Online](#)

Desasocia un método de controlador de un evento.

NOTE

Los atributos de evento en C++ nativo son incompatibles con C++ estándar. No se compilan cuando se especifica el `/permissive-` modo de cumplimiento.

Sintaxis

```
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 de eventos y `EventMethod` es el evento.
- Eventos COM: `SourceClass` es la interfaz de origen de eventos y `EventMethod` es uno de sus métodos.
- Eventos administrados: `SourceClass` es la clase de origen de eventos y `EventMethod` es el evento.

`interface`

Nombre de la interfaz que se va a desenlazar del *receptor*, solo para los receptores de eventos com en los que el parámetro `layout_dependent` del `event_receiver` atributo es `true`.

`source`

Un puntero a una instancia del origen de eventos. Dependiendo del código `type` especificado en, el `event_receiver` *origen* puede ser uno de estos tipos:

- Un puntero nativo de objeto de origen de eventos.
- `IUnknown` Puntero basado en (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 al mismo; Si no se especifica

el nombre de clase, `__unhook` se supone que la clase es aquella en la que se llama.

- Eventos de C++ nativo: `ReceiverClass` es la clase de 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 de receptor de eventos y `HandlerMethod` es el controlador.

`receiver` opta Un puntero a una instancia de la clase de receptor de eventos. Si no se especifica un receptor, el valor predeterminado es la clase o estructura del receptor en la que `__unhook` se llama a.

Uso

Puede usarse en cualquier ámbito de función, incluido `main`, fuera de la clase de receptor de eventos.

Observaciones

Utilice 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 el segundo formulario (dos argumentos) de `__unhook` solo para un receptor de eventos com; 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 llama a `__unhook` en un evento y un controlador de eventos que todavía no están enlazados, no tendrá ningún 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 para los eventos com.

Una alternativa al uso de `__unhook` es usar el operador-=.

Para obtener información sobre la codificación de eventos administrados en la nueva sintaxis, vea [Event](#).

NOTE

Una clase o struct basada en plantilla no puede contener eventos.

Ejemplo

Vea [control de eventos en C++ nativo](#) y [control de eventos en com](#) para obtener ejemplos.

Vea también

Palabra

`event_source`
`event_receiver`

[__event](#)

[__hook](#)

[__raise](#)

Control de eventos en C++ nativo

06/03/2021 • 3 minutes to read • [Edit Online](#)

En el control de eventos de C++ nativo, se configura 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 que las clases a las que se aplican desencadenen eventos y controlen eventos en un contexto no COM nativo.

NOTE

Los atributos de evento en C++ nativo son incompatibles con C++ estándar. No se compilan cuando se especifica el `/permissive-` modo de cumplimiento.

Declarar eventos

En una clase de origen de eventos, use la `_event` palabra clave 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, porque el compilador 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

En una clase de receptor de eventos, se definen los controladores de eventos. Los controladores de eventos son métodos con firmas (tipos de valor devuelto, convenciones de llamada y argumentos) que coinciden con el evento que controlarán.

Enlazar controladores de eventos a eventos

Además, en una clase de receptor 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 de origen del evento. 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 está desactivada `/permissive-`.

Ejemplo

Código

```

// 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);
}

```

Salida

```

MyHandler2 was called with value 123.
MyHandler1 was called with value 123.

```

Vea también

[Control de eventos](#)

Control de eventos en COM

06/03/2021 • 7 minutes to read • [Edit Online](#)

En el control de eventos COM, se configura un origen de eventos y un receptor de eventos mediante los `event_source` `event_receiver` atributos y, respectivamente, especificando `type = com`. Estos atributos insertan el código adecuado para las interfaces personalizadas, de envío y duales. El código insertado permite que las clases con atributos desencadenen eventos y controlen los eventos a través de los puntos de conexión COM.

NOTE

Los atributos de evento en C++ nativo son incompatibles con C++ estándar. No se compilan cuando se especifica el `/permissive-` modo de cumplimiento.

Declarar eventos

En una clase de origen de eventos, use la `_event` palabra clave en una declaración de interfaz para declarar los métodos de la 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 estar todos *en* parámetros). El tipo de valor devuelto puede ser void o cualquier tipo entero.

Definir controladores de eventos

Los controladores de eventos se definen en una clase de receptor de eventos. Los controladores de eventos son métodos con firmas (tipos de valor devuelto, convenciones de llamada y argumentos) que coinciden con el evento que controlarán. En el caso de los eventos COM, no es necesario que las convenciones de llamada coincidan. Para obtener más información, vea [eventos com dependientes del diseño](#) a continuación.

Enlazar controladores de eventos a eventos

Además, en una clase de receptor 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.

NOTE

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 `embedded_idl` calificador de importación, de modo que la biblioteca de tipos de origen de eventos se escriba en el archivo. TLH con el código generado por el atributo conservado.

Desencadenar eventos

Para desencadenar un evento, llame a un método en la interfaz declarada con la `_event` palabra clave en la clase de origen del evento. Si se han enlazado controladores al evento, se llamará a los controladores.

Código de evento 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.

```

// 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:

```

// 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:

```

// 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

```

MyHandler1 was called with value 123.
MyHandler2 was called with value 123.

```

Eventos COM dependientes del diseño

La dependencia de diseño solo es un problema para la programación COM. En el control de eventos nativo y

administrado, las firmas (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 COM, cuando se establece el `Layout_dependent` parámetro de `event_receiver` en `true`, se aplica el nombre y la coincidencia de la firma. Los nombres y las firmas de los controladores del receptor de eventos y de los eventos enlazados 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 coincidir entre el método de evento desencadenador y los métodos de enlace (sus delegados). Es ligeramente más eficaz `Layout_dependent = true`.

Suponga, por ejemplo, que se define `IEventSource` para que tenga los siguientes métodos:

```
[id(1)] HRESULT MyEvent1([in] int value);
[id(2)] HRESULT MyEvent2([in] int value);
```

Suponga que el origen de eventos tiene el siguiente formato:

```
[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:

```
[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
    }
};
```

Vea también

[Control de eventos](#)

Modificadores específicos de Microsoft

06/03/2021 • 2 minutes to read • [Edit Online](#)

En esta sección se describen las extensiones específicas de Microsoft para C++ en las áreas siguientes:

- [Direccionamiento basado en](#), 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 __declspec (Restrict), 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
__unaligned	Especifica que un puntero a un tipo u otros datos no esté alineado.	No

PALABRA CLAVE	SIGNIFICADO	¿SE USA PARA FORMAR TIPOS DERIVADOS?
<code>_vectorcall</code>	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

06/03/2021 • 2 minutes to read • [Edit Online](#)

Esta sección contiene los siguientes temas:

- [Gramática __based](#)
- [Punteros con base](#)

Consulta también

[Modificadores específicos de Microsoft](#)

Gramática __based

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 las compilaciones de 32 bits y 64 bits es "basada en un puntero" que define un tipo que contiene un desplazamiento de 32 bits o de 64 bits a una base de 32 bits o de 64 bits, o en función de `void`.

Gramática

modificador-de-intervalo: __based (base-expresión)

base-expresión: based-variablebased-Abstract-declaratorsegment-namesegment-Cast

based-variable: Identifier

based-Abstract-declarator: abstract-declarator

tipo base: type-name

FIN de Específicos de Microsoft

Vea también

[Punteros con base](#)

Punteros con base (C++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

La `__based` palabra clave permite declarar punteros basados en punteros (punteros que son desplazamientos de punteros existentes). La `__based` palabra clave es específica de Microsoft.

Sintaxis

```
type __based( base ) declarator
```

Observaciones

Los punteros basados en direcciones de puntero son la única forma de la `__based` palabra clave válida en compilaciones de 32 bits o de 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:

```
// 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`.

NOTE

Los identificadores persistentes que contienen punteros también se pueden realizar 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 un sinónimo de `__based` a menos que se especifique la opción del compilador [/za \(Disable Language Extensions\)](#).

Ejemplo

El código siguiente muestra cómo se cambia un puntero basado mediante el cambio de su base.

```
// 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;
}
```

```
1
2
10
11
```

Consulta también

[Palabras clave](#)
[alloc_text](#)

Convenciones de llamada

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 procesadores x64, vea [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](#)

Consulta también

[Modificadores específicos de Microsoft](#)

Paso de argumentos y convenciones de nomenclatura

06/03/2021 • 3 minutes to read • [Edit Online](#)

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 las 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.

NOTE

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 cómo definir su propio código de prólogo y epílogo 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 Visual C++ arm](#).

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/a	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

PALABRA CLAVE	LIMPIEZA DE LA PILA	PASO DE PARÁMETROS
<code>_thiscall</code>	Destinatario	Insertado en la pila; <code>this</code> puntero 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`

06/03/2021 • 3 minutes to read • [Edit Online](#)

`__cdecl` es la Convención de llamada predeterminada para los programas de C y C++. Dado que el llamador limpia la pila, puede realizar `vararg` funciones. La `__cdecl` Convención de llamada crea archivos ejecutables mayores que `__stdcall`, porque requiere que cada llamada a función incluya el código de limpieza de la pila. En la lista siguiente se muestra la implementación de esta convención de llamada. El `__cdecl` modificador 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 (<u>) se antepone a los nombres, excepto cuando _ se exportan <i>cdecl funciones que usan la vinculación C</i>.</u>
Convención de traducción de mayúsculas y minúsculas	No se lleva a cabo una traducción de mayúsculas y minúsculas.

NOTE

Para obtener información relacionada, vea [nombres representativos](#).

Coloque el `__cdecl` modificador delante de una variable o un nombre de función. Dado que las convenciones de llamada y nomenclatura de C son el valor predeterminado, la única vez que debe usar `__cdecl` en el código x86 es cuando se ha especificado la `/Gv` `/Gz` opción del compilador (vectorcall), (Stdcall) o `/Gr` (fastcall). La opción del compilador `/GD` fuerza la `__cdecl` Convención de llamada.

En los procesadores ARM y x64, `__cdecl` se acepta, pero el compilador normalmente lo omite. 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 x64 predeterminada.

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:

```
struct CMyClass {  
    void __cdecl mymethod();  
};
```

esto:

```
void CMyClass::mymethod() { return; }
```

equivale a esto:

```
void __cdecl CMyClass::mymethod() { return; }
```

Por compatibilidad con versiones anteriores, **Cdecl** y **_cdecl** son un 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**.

```
// 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, ...);
```

Consulta también

[Paso de argumentos y convenciones de nomenclatura](#)

[Palabras clave](#)

`__clrcall`

06/03/2021 • 6 minutes to read • [Edit Online](#)

Especifica que solo se puede llamar a una función desde código administrado. Use `__clrcall` para todas las funciones virtuales a las que solo se llamará desde el 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.

Use `__clrcall` para mejorar el rendimiento al llamar desde una función administrada a una función administrada virtual o desde una función administrada a una función administrada a través de un 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. Al marcar una función como `__clrcall`, indica que la implementación de la función debe ser MSIL y que no se generará la función de punto de entrada nativo.

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 está administrada y no es necesario pasar por la transición de administrado a nativo. En ese caso, el compilador usa el punto de entrada administrado.

Cuando `/clr` `/clr:pure` se usa (no o `/clr:safe`) y no se utiliza `__clrcall`, tomar la dirección de una función siempre devuelve la dirección de la función de punto de entrada nativo. Cuando se utiliza `__clrcall`, no se crea la función de punto de entrada nativo, por lo que se obtiene la dirección de la función administrada, no una función de thunk de punto de entrada. Para obtener más información, vea [double thunk](#). 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 de la operación de compilación se marque como algo que no sea `__clrcall`. Cuando se usa `/clr: Pure`, solo se puede especificar `__clrcall` en punteros a función y declaraciones externas.

Puede llamar directamente a las funciones de `__clrcall` desde el código de C++ existente que se compiló mediante `/CLR` siempre que esa función tenga una implementación de MSIL. no se puede llamar a `__clrcall` funciones directamente desde funciones que tienen ASM en línea y llamar a intrínsecos específicos de la CPU, por ejemplo, incluso si esas funciones se compilan con `/clr`.

`__clrcall` los punteros de función solo están diseñados para usarse en el dominio de aplicación en el que se crearon. En lugar de pasar `__clrcall` punteros de función entre dominios de aplicación, use [CrossAppDomainDelegate](#). Para obtener más información, consulte [dominios de aplicación y Visual C++](#).

Ejemplos

Tenga en cuenta que cuando una función se declara con `__clrcall`, se generará código cuando sea necesario. por ejemplo, cuando se llama a la función.

```

// clrcall1.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();
}

```

```

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).

```

// clrcall3.cpp
// compile with: /clr
void Test() {
    System::Console::WriteLine("in Test");
}

int main() {
    void (*pTest)() = &Test;
    (*pTest)();

    void (__clrcall *pTest2)() = &Test;
    (*pTest2)();
}

```

Consulta también

[Paso de argumentos y convenciones de nomenclatura](#)

[Palabras clave](#)

`__stdcall`

06/03/2021 • 3 minutes to read • [Edit Online](#)

La `__stdcall` Convención de llamada se usa para llamar a funciones de la API de Win32. El destinatario limpia la pila, por lo que el compilador realiza `vararg` funciones `__cdecl`. Las funciones que usan esta convención de llamada requieren un prototipo de función. El `__stdcall` modificador es específico de Microsoft.

Sintaxis

```
tipo de valor devuelto __stdcall function-Name[ ( lista de argumentos ) ]
```

Observaciones

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 carácter de subrayado (<code>_</code>) tiene como prefijo el nombre. El nombre va seguido del signo de arroba (<code>@</code>) seguido del número de bytes (en formato 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.

Por 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 `__stdcall` modificador devuelven los valores de la misma manera que las funciones declaradas mediante `__cdecl`.

En los procesadores ARM y x64, `__stdcall` el compilador acepta y omite; 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,

```
struct CMyClass {  
    void __stdcall mymethod();  
};
```

this

```
void CMyClass::mymethod() { return; }
```

equivale a esto

```
void __stdcall CMyClass::mymethod() { return; }
```

Ejemplo

En el ejemplo siguiente, el uso de los `__stdcall` resultados en todos los `WINAPI` tipos de función se administran como una llamada estándar:

```
// 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, ...);
```

Consulta también

[Paso de argumentos y convenciones de nomenclatura](#)

[Palabras clave](#)

__fastcall

06/03/2021 • 4 minutes to read • [Edit Online](#)

Específicos de Microsoft

La `__fastcall` Convención de llamada especifica que los argumentos de las funciones se van a pasar 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 (@) tiene el prefijo en los nombres; un signo de arroba seguido del número de bytes (en formato decimal) en la lista de parámetros tiene como sufijo los nombres.
Convención de traducción de mayúsculas y minúsculas	No se lleva a cabo una traducción de mayúsculas y minúsculas.

NOTE

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` a menos que la función se declare con un atributo en conflicto o el nombre de la función sea `main`.

Los `__fastcall` compiladores que tienen como destino las arquitecturas ARM y x64 aceptan y omiten la palabra clave; en un chip x64, por Convención, los primeros cuatro argumentos se pasan en registros cuando es posible y los argumentos adicionales se pasan en la pila. Para obtener más información, consulte la [Convención de Llamada 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:

```
struct CMyClass {  
    void __fastcall mymethod();  
};
```

esto:

```
void CMyClass::mymethod() { return; }
```

equivale a esto:

```
void __fastcall CMyClass::mymethod() { return; }
```

Por compatibilidad con versiones anteriores, `_fastcall` es un sinónimo de `__fastcall` a menos que se especifique la opción del compilador [/za \(Disable Language Extensions\)](#).

Ejemplo

En el siguiente ejemplo, se pasan argumentos a la función `DeleteAggrWrapper` en los registros:

```
// 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

Vea también

[Paso de argumentos y convenciones de nomenclatura](#)

[Palabras clave](#)

La Convención de llamada **específica de Microsoft** `_thiscall` se usa en las funciones miembro de clase de C++ en la arquitectura x86. Es la Convención de llamada predeterminada utilizada por las funciones miembro que no usan argumentos variables (`vararg` funciones).

En `_thiscall`, el destinatario limpia la pila, lo que es imposible para `vararg` las funciones. Los argumentos se insertan en la pila de derecha a izquierda. El `this` puntero se pasa a través de la ECx de registro y no en la pila.

En las máquinas ARM, ARM64 y x64, `_thiscall` el compilador acepta y omite. Esto se debe a que usa de forma predeterminada una Convención de llamada basada en registros.

Una razón para usar `_thiscall` es en las clases cuyas funciones miembro usan `_clrcall` de forma predeterminada. En ese caso, puede usar para hacer que se pueda `_thiscall` llamar a funciones de miembro individuales 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 `/clr:pure` `/clr:safe` Opciones del compilador y están en desuso en visual Studio 2015 y no se admiten en visual Studio 2017.

`vararg` las funciones miembro usan la `_cdecl` Convención de llamada. Todos los argumentos de función se insertan en la pila, y el `this` puntero se coloca en la pila en último lugar.

Dado que esta Convención de llamada solo se aplica a C++, no tiene un 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 tiene que volver a especificarla 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

06/03/2021 • 21 minutes to read • [Edit Online](#)

Específicos de Microsoft

La `__vectorcall` Convención de llamada especifica que los argumentos de las funciones se van a pasar en registros, siempre que sea posible. `__vectorcall` utiliza más registros para argumentos que `__fastcall` o la Convención de [Llamada x64](#) predeterminada que se usa. La `__vectorcall` Convención de llamada solo se admite en código nativo en procesadores x86 y x64 que incluyen extensiones SIMD de streaming 2 (sse2) y versiones posteriores. `__vectorcall` Se usa para acelerar funciones que pasan varios argumentos de vector de punto flotante o SIMD y realizan operaciones que aprovechan los argumentos cargados en registros. La lista siguiente muestra las características que son comunes a 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	Los nombres de función tienen como sufijo dos signos "arroba" (@ @) seguidos por el número de bytes (en formato decimal) en la lista de parámetros.
Convención de traducción de mayúsculas y minúsculas	No se lleva a cabo la traducción de mayúsculas y minúsculas.

El uso de la `/Gv` opción del compilador 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 en conflicto, use una `vararg` lista de argumentos de variable o tenga el nombre `main`.

Puede pasar tres tipos de argumentos por registro en `__vectorcall` las funciones: 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 un equipo x86 u 8 bytes en un equipo x64) y se puede convertir en un entero de longitud de registro y viceversa sin cambiar su representación de bits. Por ejemplo, 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 volver a su tipo original sin cambiar es un tipo entero. Los tipos enteros incluyen el puntero, la referencia y los `struct` `union` tipos de 4 bytes (8 bytes en x64) o menos. En las plataformas x64, `struct` `union` los tipos y mayores se pasan por referencia a la memoria asignada por el llamador; en las plataformas x86, se pasan por valor en la pila.

Un tipo de vector es un tipo de punto flotante, por ejemplo, o `float`, `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 de HVA `struct` que contiene tres tipos de vector idénticos y tiene una alineación de 32 bytes:

```
typedef struct {
    __m256 x;
    __m256 y;
    __m256 z;
} hva3; // 3 element HVA type on __m256
```

Declare las funciones explícitamente con la `__vectorcall` palabra clave en los archivos de encabezado para permitir que el código compilado por separado se vincule sin errores. Las funciones deben ser prototipo para utilizar `__vectorcall` y no pueden utilizar una `vararg` lista de argumentos de longitud variable.

Una función miembro se puede declarar con el `__vectorcall` especificador. El puntero oculto lo `this` pasa el registro como el primer argumento de tipo entero.

En los equipos ARM, `__vectorcall` el compilador acepta y omite.

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:

```
struct MyClass {  
    void __vectorcall mymethod();  
};
```

esto:

```
void MyClass::mymethod() { return; }
```

equivale a esto:

```
void __vectorcall MyClass::mymethod() { return; }
```

El `__vectorcall` modificador de Convención de llamada debe especificarse cuando se crea un puntero a una `__vectorcall` función. En el ejemplo siguiente se crea un `typedef` para un puntero a una `__vectorcall` función que toma cuatro `double` argumentos y devuelve un `_m256` valor:

```
typedef _m256 (__vectorcall * vcfnptr)(double, double, double, double);
```

Por compatibilidad con versiones anteriores, `_vectorcall` es un sinónimo de `__vectorcall` a menos que se especifique la opción del compilador `/za` ([deshabilitar extensiones de lenguaje](#)).

Convención `__vectorcall` en x64

La `__vectorcall` Convención de llamada en x64 amplía la Convención de llamada x64 estándar para aprovechar registros adicionales. 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 `this` puntero oculto se trata como el primer argumento de tipo entero. Cuando un argumento de HVA de uno de los cuatro primeros argumentos no se puede pasar en los registros disponibles, se pasa en su lugar una referencia a la memoria asignada por el Llamador en el registro correspondiente de tipo entero. 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 `_m256` tipos 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 la pila de sombra asignado para los argumentos de tipo de vector se fija en 8 bytes y la opción no se [/homeparams](#) 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 no utilizados XMM0 a XMM5 (o YMM0 a YMM5, para `_m256` los tipos), siempre y cuando 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 `__vectorcall` las funciones se devuelven por valor en registros cuando 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 la 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:

```
// 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 `__vectorcall` Convención de llamada sigue la `__fastcall` Convención de los argumentos de tipo entero de 32 bits y aprovecha los registros vectoriales de SSE para los argumentos de tipo de vector y HVA.

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 `this` puntero 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 `_m256` tipos 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 del 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 de vector no utilizados XMM0 a XMM5 (o YMM0 a YMM5, para `_m256` tipos), siempre y cuando 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 `__vectorcall` las funciones se devuelven por valor en registros cuando 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 llamador, y la función llamada borra la pila justo antes de que se devuelva. Solo los argumentos que no se colocan en registros se insertan en la pila.

Ejemplos:

```
// 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);
}

```

Finalizar específico de Microsoft

Consulta también

[Paso de argumentos y convenciones de nomenclatura](#)
[Palabras clave](#)

Ejemplo de llamada: Prototipo de función y llamada

06/03/2021 • 2 minutes to read • [Edit Online](#)

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.

```
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 del ejemplo de llamada](#).

FIN de Específicos de Microsoft

Vea también

[Convenciones de llamada](#)

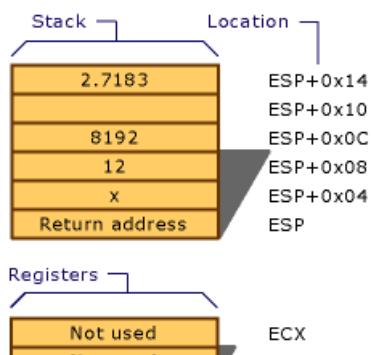
Resultados del ejemplo de llamada

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

`_cdecl`

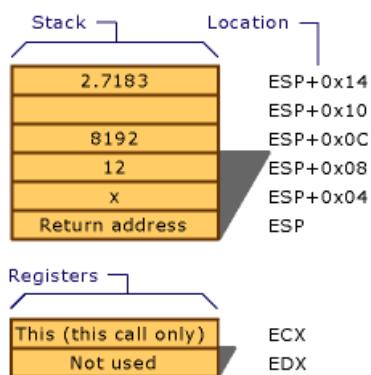
El nombre de la función decorada de C es `_MyFunc`.



`_cdecl` Convención de llamada

`_stdcall` y `thiscall`

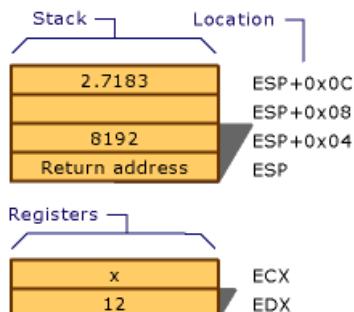
El nombre representativo de C (`_stdcall`) es `_MyFunc@20`. El nombre representativo de C++ es específico de la implementación.



Las convenciones de llamada de `_stdcall` y `thiscall`

`_fastcall`

El nombre representativo de C (`_fastcall`) es `@MyFunc@20`. El nombre representativo de C++ es específico de la implementación.



La convención de llamada __fastcall

FIN de Específicos de Microsoft

Vea también

[Ejemplo de llamada: prototipo de función y llamada](#)

Llamadas de función naked

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Las funciones declaradas con el `naked` atributo se emiten sin código de prólogo o epílogo, lo que le permite escribir sus propias secuencias de prólogo/epílogo personalizadas 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

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Las reglas y las limitaciones siguientes se aplican a las funciones naked:

- La `return` instrucción 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.
- La `naked` palabra clave se omite al compilar con [/CLR](#).
- En el caso de las funciones de `__fastcall` Naked, siempre que haya una referencia en el código de C/C++ a uno de los argumentos de registro, el código de prólogo debe almacenar los valores de que se registran en la ubicación de la pila para esa variable. Por ejemplo:

```
// 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

Vea también

[Llamadas a funciones Naked](#)

Consideraciones para escribir código de prólogo/epílogo

06/03/2021 • 3 minutes to read • [Edit Online](#)

Específicos de Microsoft

Antes de escribir sus propias secuencias de código de prólogo y epílogo, es importante comprender cómo se diseña el marco de pila. También es útil saber cómo usar el `__LOCAL_SIZE` símbolo.

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
```

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 de 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` solo 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 epílogo se usa el `__LOCAL_SIZE` símbolo en la secuencia de prólogo:

```
// 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
    }
}
```

FIN de Específicos de Microsoft

Vea también

[Llamadas a funciones Naked](#)

Coprocesador de punto flotante y convenciones de llamada

06/03/2021 • 2 minutes to read • [Edit Online](#)

Si está escribiendo 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 esté devolviendo un `float` `double` valor o (que la función debe devolver en St (0)).

Vea también

[Convenciones de llamada](#)

Convenciones de llamada obsoletas

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Ya no se admiten las convenciones de llamada de `__pascal`, `__fortran` y `__syscall`. Puede emular su funcionalidad mediante una de las convenciones de llamada admitidas y las opciones del vinculador adecuadas.

`<windows.h>` ahora admite la macro WINAPI, que se traduce en la Convención de llamada adecuada para el destino. Use WINAPI donde anteriormente usó PASCAL o `__far __pascal`.

FIN de Específicos de Microsoft

Vea también

[Paso de argumentos y convenciones de nomenclatura](#)

restrict (C++ AMP)

06/03/2021 • 4 minutes to read • [Edit Online](#)

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).

NOTE

Para obtener información sobre la `restrict` palabra clave que forma parte de los `__declspec` atributos de clase de almacenamiento, vea [Restrict](#).

La `restrict` cláusula tiene las siguientes formas:

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> .

Observaciones

La `restrict` palabra clave 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 `restrict` cláusula es igual que una función que tiene la `restrict(cpu)` cláusula.

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 `int` las `unsigned int` variables, `float` y `double`, y las clases y estructuras que solo contienen estos tipos. `bool` también se permite, pero debe tener una alineación 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.
 - `goto` afirma.
 - Instrucciones con etiqueta.
 - `try **** catch` instrucciones, o `throw`.
 - Variables globales.
 - Variables estáticas. En su lugar, use [Típico_static palabra clave](#).
 - `dynamic_cast` conversiones.
 - `typeid` Operador.
 - Declaraciones asm.
 - Varargs.

Para obtener una explicación de las limitaciones de las funciones, consulte [restricciones de restricción \(amp\)](#).

Ejemplo

En el ejemplo siguiente se muestra cómo utilizar la `restrict(amp)` cláusula.

```
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();
}
```

Consulta también

[C++ AMP \(C++ Accelerated Massive Parallelism\)](#)

tile_static (Palabra clave)

06/03/2021 • 4 minutes to read • [Edit Online](#)

La palabra clave `tile_static` se usa 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, vea [uso de mosaicos](#).

La palabra clave `tile_static` tiene las siguientes limitaciones:

- 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 se declara una variable de `tile_static` en un gráfico de llamadas cuya raíz es una llamada no en mosaico a `parallel_for_each`, se genera una advertencia y el comportamiento de la variable no está definido.

Ejemplo

En el ejemplo siguiente se muestra cómo se puede utilizar una variable de `tile_static` para acumular datos en varios subprocessos de un mosaico.

```
// 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  
    }  
)
```

```

// 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,
    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
```

Consulta también

[Modificadores específicos de Microsoft](#)

[Información general de C++ AMP](#)

[parallel_for_each \(Función\) \(C++ AMP\)](#)

[Tutorial: multiplicación de matrices](#)

Específicos de Microsoft

La sintaxis de atributo extendido para especificar información de clase de almacenamiento utiliza la `__declspec` palabra clave, que especifica que una instancia de un tipo determinado se almacenará con un atributo de clase de almacenamiento específico de Microsoft que se enumera a continuación. Algunos ejemplos de otros modificadores de clase de almacenamiento incluyen las `static` `extern` palabras clave y. 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( ***número de )  
    allocate(" segname ")  
    allocator  
    appdomain  
    code_seg(" segname ")  
    deprecated  
    dllimport  
    dllexport  
    jitintrinsic  
    naked  
    noalias  
    noinline  
    noreturn  
    noexcept  
    novtable  
    no_SANITIZE_ADDRESS  
    process  
    property( { get= Get-FUNC-Name| ,put= 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: `, , , , „ , , , , , , , , , , , , , , , , y. También admite estos atributos de objetos COM: y uuid.`

Los `code_seg` `dllexport` atributos de clase de almacenamiento, `dllimport`, `naked` „„„ `noalias` `nothrow`, `no-sanitize_address`, `property`, `restrict`, `selectany`, `thread` y `uuid` son propiedades solo de la declaración del objeto o la función a la que se aplican. El `thread` atributo solo afecta a los datos y objetos. Los `naked`, `spectre` atributos y solo afectan a las funciones. Los `dllimport` `dllexport` atributos y afectan a funciones, datos y objetos. Los `property` `selectany` atributos, y `uuid` afectan a objetos com.

Por 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 `_declspec` palabras clave deben colocarse al principio de una declaración simple. El compilador omite, sin advertencia, las `_declspec` palabras clave colocadas después de * o & y delante del identificador de variable en una declaración.

Un `_declspec` atributo especificado al principio de una declaración de tipos definidos por el usuario se aplica a la variable de ese tipo. Por ejemplo:

```
_declspec(dllexport) class X {} varX;
```

En este caso, el atributo se aplica a `varX`. Un `_declspec` atributo colocado después de `class` la `struct` palabra clave o se aplica al tipo definido por el usuario. Por ejemplo:

```
class _declspec(dllexport) X {};
```

En este caso, el atributo se aplica a `X`.

La regla general para utilizar el `_declspec` atributo para las declaraciones simples es la siguiente:

decl-Specifier-SEQ init-declarator-List;

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 `_declspec` extensión. *Init-declarator-List* debe contener, entre otras cosas, la parte del puntero de las declaraciones. Por ejemplo:

```
_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:

```
// Example of the _declspec keyword
__declspec( thread ) int tls_i = 1;
```

FIN de Específicos de Microsoft

Vea también

Palabra

Atributos extendidos de clase de almacenamiento de C

align (C++)

06/03/2021 • 15 minutes to read • [Edit Online](#)

En Visual Studio 2015 y versiones posteriores, use el especificador estándar de C++ 11 `para controlar la alineación. Para obtener más información, vea alignment.`

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

```
declarador __declspec (Align ( #))
```

Observaciones

Las aplicaciones de escritura que utilizan las últimas instrucciones de procesador presentan nuevas restricciones y problemas. Muchas instrucciones nuevas requieren datos que se alineen con límites de 16 bytes. Además, al alinear los datos usados con frecuencia en el tamaño de la línea de caché del procesador, se mejora el rendimiento de la memoria caché. Por ejemplo, si define una estructura cuyo tamaño es inferior a 32 bytes, es posible que desee una alineación de 32 bytes para asegurarse de que los objetos de ese tipo de estructura se almacenan en caché de forma eficaz.

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 son los datos que se están declarando como alineados.

Para obtener información sobre cómo devolver un valor de tipo `size_t` que es el requisito de alineación del tipo, vea [alignof](#). Para obtener información sobre cómo declarar punteros no alineados al establecer como destino procesadores de 64 bits, vea [unaligned](#).

Puede usar `__declspec(align(#))` cuando defina `struct`, `union` o `class`, o cuando declare una variable.

El compilador no garantiza ni intenta conservar el atributo de alineación de los datos durante una operación de copia o transformación de datos. Por ejemplo, `memcpy` puede copiar un struct declarado con `__declspec(align(#))` en cualquier ubicación. Los asignadores ordinarios (por ejemplo, `malloc`, C++ `operator new` y los asignadores de Win32) normalmente devuelven memoria que no está suficientemente alineada para `__declspec(align(#))` estructuras 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 función. Al pasar datos que tienen un atributo `Alignment` por valor en la pila, la alineación se controla mediante 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.

Si no se usa `__declspec(align(#))`, el compilador generalmente alinea los datos en los límites naturales según el procesador de destino y el tamaño de los datos, hasta los límites de 4 bytes en los procesadores de 32 bits y los límites de 8 bytes en procesadores de 64 bits. Los datos de las clases o estructuras se alinean en la clase o estructura en el mínimo de su alineación natural y el valor de empaquetado actual (de `#pragma pack` o la `/Zp` opción del compilador).

Este ejemplo muestra el uso de `__declspec(align(#))`:

```
__declspec(align(32)) struct Str1{
    int a, b, c, d, e;
};
```

Ahora, este tipo tiene un atributo de alineación de 32 bytes. Significa que todas las instancias estáticas y automáticas se inician en un límite de 32 bytes. Los tipos de estructura adicionales declarados con este tipo como un miembro conservan el atributo de alineación de este tipo, es decir, cualquier estructura con `Str1` como un elemento tiene un atributo de alineación de al menos 32.

Aquí, `sizeof(struct Str1)` es igual a 32. Implica que, si se crea una matriz de `str1` objetos y la base de la matriz tiene una alineación de 32 bytes, cada miembro de la matriz también tiene 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 `sizeof` valor de 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 miembros o del valor de alineación de estructura completo, 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 Example](#)
- [Definir 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 estructura \(específico de x64\)](#)

Ejemplos de align

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:

```
#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 `int` miembro requiere una alineación de 4 bytes, pero la alineación de la propia estructura se declara como 32. La alineación general es 32.

```
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).

```
__declspec(align(8)) struct S2 {
    int a, b, c, d;
};
```

En el ejemplo siguiente, `sizeof(struct S3)` devuelve 64.

```
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 siguientes `a : s1` empieza en el desplazamiento 32. `S4` hereda el requisito de alineación de `s1`, porque es el requisito de alineación mayor de la estructura. `sizeof(struct S4)` devuelve 64.

```
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 `sizeof` valor de cada miembro de la matriz no se ve afectado cuando se usa `__declspec(align(#))`.

```
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:

```
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:

```

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.

```

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 inferior a arg . Por ejemplo, si usa `malloc(7)`, la alineación es de 4 bytes.

Definir 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 esta manera:

```

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 de subprocessos

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 subprocesso.

```

// 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 `/zp` opción del compilador y la `pack` pragma tienen el efecto de empaquetar datos para miembros de estructura y de Unión. En este ejemplo se muestra cómo `/zp` y `__declspec(align(#))` trabajan juntos:

```

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 en `/zp` valores distintos (o `#pragma pack`), que muestra cómo interactúan los dos.

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, vea [/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

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

El `allocate` especificador de declaración nombra un segmento de datos en el que se asignará el elemento de datos.

Sintaxis

```
* __declspec(allocate(" ***segname )) declarador
```

Observaciones

El nombre *segname* se debe declarar mediante una de las siguientes pragmas:

- `code_seg`
- `const_seg`
- `data_seg`
- `init_seg`
- `transversal`

Ejemplo

```
// allocate.cpp
#pragma section("mycode", read)
__declspec(allocate("mycode")) int i = 0;

int main() {
```

FIN de Específicos de Microsoft

Vea también

[__declspec](#)

[Palabras clave](#)

Específicos de Microsoft

El `allocator` especificador de declaración se puede aplicar a funciones de asignación de memoria personalizadas para que las asignaciones estén visibles a través del seguimiento de eventos para Windows (ETW).

Sintaxis

```
__declspec(allocator)
```

Observaciones

El generador de perfiles de memoria nativa en Visual Studio funciona mediante la recopilación de datos de eventos ETW de asignación emitidos por durante el tiempo de ejecución. Los asignadores de CRT y Windows SDK se han anotado en el nivel de origen para que se pueden capturar los datos de asignación. Si escribe sus propios asignadores, las funciones que devuelven un puntero a la memoria de montón recién asignada se pueden decorar con `__declspec(allocator)`, tal como se puede observar en este ejemplo para `myMalloc`:

```
__declspec(allocator) void* myMalloc(size_t size)
```

Para obtener más información, vea [medir el uso de memoria en Visual Studio](#) y [eventos de montón ETW nativos personalizados](#).

FIN de Específicos de Microsoft

appdomain

06/03/2021 • 4 minutes to read • [Edit Online](#)

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. Vea [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 en [/CLR](#). Las opciones del compilador `/clr: Pure` y `/clr: Safe` están en desuso en Visual Studio 2015 y no se admiten en Visual Studio 2017.

`__declspec(appdomain)` solo es válido 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 de subprocessos \(TLS\)](#). Los subprocessos 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.

Existen limitaciones para mezclar el uso de las variables por proceso y por AppDomain; Vea el [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 específico, vea [Call_in_appdomain función](#).

Ejemplo

```
// 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");
    }
}
```

```

(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::CurrentDomain;
    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);
}

```

```
__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](#)

code_seg (_declspec)

06/03/2021 • 6 minutes to read • [Edit Online](#)

Específicos de Microsoft

El atributo de declaración **code_seg** nombra un segmento de texto ejecutable en el archivo. obj en el que se almacenará el código objeto de las funciones miembro de la función o clase.

Sintaxis

```
_declspec(code_seg("segname")) declarator
```

Observaciones

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 de datos con nombre en un archivo. obj que se carga en la memoria como una unidad. Un *segmento de texto* es un segmento que contiene código ejecutable. La *sección* term se usa a menudo indistintamente con Segment.

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. No es `segname` necesario especificar el nombre en una pragma de `sección` antes de que se pueda usar en una declaración. De forma predeterminada, cuando no se especifica `code_seg`, el código objeto se coloca en un segmento denominado .text. Un atributo de `code_seg` invalida cualquier directiva de `code_seg de #pragma` existente. Un atributo `code_seg` aplicado a una función miembro reemplaza cualquier atributo `code_seg` aplicado a la clase envolvente.

Si una entidad tiene un atributo `code_seg`, todas las declaraciones y definiciones de la misma entidad deben tener atributos `code_seg` idénticos. Si una clase base tiene un atributo `code_seg`, las clases derivadas deben tener el mismo atributo.

Cuando se aplica un atributo `code_seg` a una función de ámbito de espacio de nombres o a una función miembro, el código 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 (esto incluye las funciones miembro especiales generadas por el compilador) se colocan en el segmento especificado. Las clases definidas localmente (por ejemplo, las clases definidas en el cuerpo de una función miembro) no heredan el atributo `code_seg` del ámbito de inclusión.

Cuando se aplica un atributo `code_seg` a una clase de plantilla o una función de plantilla, 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 de `code_seg` o otro diferente en la especialización. No se puede aplicar un atributo `code_seg` 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 segmento .text. La directiva `#pragma code_seg` no invalida este valor predeterminado. Use el atributo `code_seg` en la clase, plantilla de clase o plantilla de función para controlar dónde se coloca el código generado por el compilador.

Las lambdas heredan **code_seg** atributos de su ámbito de inclusión. Para especificar un segmento para una expresión lambda, aplique un **code_seg** atributo después de la cláusula de declaración de parámetros y antes de cualquier especificación mutable o de excepción, cualquier especificación de tipo de valor devuelto final y el cuerpo de la expresión lambda. Para obtener más información, vea [Sintaxis de expresiones lambda](#). En este ejemplo se define una expresión lambda en un segmento denominado PagedMem:

```
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. Si se define una función virtual en una clase derivada que reside en un segmento paginado cuando el método de la clase base reside en un segmento no paginado, otros métodos de clase base o el código de usuario puede suponer que al invocar el método virtual no se desencadenará un error de página.

Ejemplo

En este ejemplo se muestra cómo un atributo **code_seg** controla la selección de ubicación de segmentos cuando se usa la especialización de plantilla implícita y explícita:

```

// 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

Vea también

[__declspec](#)

[Palabras clave](#)

en desuso (C++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

Este tema trata sobre la declaración de declspec obsoleto específica de Microsoft. Para obtener información sobre el atributo de C++ 14 `[[deprecated]]` e instrucciones sobre Cuándo usar ese atributo frente al declspec o pragma específico de Microsoft, vea [atributos estándar de C++](#).

Con las excepciones que se indican a continuación, la `deprecated` declaración ofrece la misma funcionalidad que la pragma `desusada`:

- La `deprecated` Declaración permite especificar formas determinadas de sobrecargas de función como desusadas, mientras que la forma pragma se aplica a todas las formas sobrecargadas de un nombre de función.
- La `deprecated` Declaración permite especificar un mensaje que se mostrará en tiempo de compilación. El texto del mensaje puede proceder de una macro.
- Las macros solo se pueden marcar como desusadas con la `deprecated` Directiva pragma.

Si el compilador encuentra el uso de un identificador desusado o el atributo estándar `[[deprecated]]`, 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.

```
// 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.

```
// 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
}
```

Consulta también

[__declspec](#)

[Palabras clave](#)

dllexport, dllimport

06/03/2021 • 3 minutes to read • [Edit Online](#)

Específicos de Microsoft

Los `__declspec(dllimport)` y `__declspec(dllexport)` atributos de clase de almacenamiento y 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
```

Observaciones

Estos atributos definen explícitamente la interfaz de la DLL para el cliente, que puede ser el archivo ejecutable u otra DLL. La declaración de funciones como `__declspec(dllexport)` elimina la necesidad de un archivo de definición de módulo (.def), al menos con respecto a la especificación de funciones exportadas. El `__declspec(dllexport)` atributo reemplaza a 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.

`__declspec(dllexport)` de una función expone la función con su nombre representativo. Para las funciones de C++, esto incluye la eliminación de nombres. Para las funciones de C o las funciones declaradas como `extern "C"`, esto incluye la decoración específica de la plataforma que se basa en la convención de llamada. Para obtener información sobre la decoración de nombres en código C/C++, vea [nombres representativos](#). No se aplica ninguna decoración de nombre a las funciones de C exportadas o a `extern "C"` las funciones de C++ mediante la `__cdecl` Convención de llamada.

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 obtener más información, vea [Exports](#). Otra manera de exportar un nombre no representativo es usar una

```
#pragma comment(linker, "/export:alias=decorated_name")
```

 Directiva en el código fuente.

Al declarar `__declspec(dllexport)` o `__declspec(dllimport)`, debe usar la [Sintaxis de atributo extendido](#) y la `__declspec` palabra clave.

Ejemplo

```
// Example of the __declspec( dllimport ) and __declspec( dllexport ) class attributes  
__declspec( dllimport ) int i;  
__declspec( dllexport ) void func();
```

Opcionalmente, para que el código sea más legible, puede utilizar definiciones de macro:

```
#define DllImport  __declspec( dllexport )
#define DllExport   __declspec( dllimport )

DllExport void func();
DllExport int i = 10;
DllImport int j;
DllExport int n;
```

Para más información, consulte:

- [Definiciones y declaraciones](#)
- [Definir funciones insertadas de C++ con dllexport y DllImport](#)
- [Reglas y limitaciones generales](#)
- [Usar dllimport y dllexport en clases de C++](#)

FIN de Específicos de Microsoft

Vea también

[__declspec](#)

[Palabras clave](#)

Definiciones y declaraciones (C++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 que se declaran como `dllimport` o `dllexport`. Todas las declaraciones incluidas en la interfaz DLL deben especificar el `dllimport` atributo o `dllexport`. Sin embargo, la definición solo debe especificar el `dllexport` atributo. 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 `extern` palabra clave con `dllexport` para forzar una declaración; de lo contrario, se implica 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
DllImport int j;      // 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

Vea también

[dllexport](#), [dllimport](#)

Definir funciones insertadas de C++ con `dllexport` y `dllimport`

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Puede definir como alineada una función con el `dllexport` atributo. 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

Vea también

[dllexport, dllimport](#)

Reglas generales y limitaciones

06/03/2021 • 5 minutes to read • [Edit Online](#)

Específicos de Microsoft

- Si declara una función o un objeto sin el `dllimport` `dllexport` atributo o, la función o el objeto no se considera 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 hacer que la función o el objeto formen parte de la interfaz DLL, debe declarar la definición de la función u 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 `dllexport` atributo, 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 `dllimport` `dllexport` declaraciones y para la misma función u objeto, el `dllexport` atributo tiene prioridad sobre el `dllimport` atributo. Sin embargo, se genera una advertencia del compilador. Por ejemplo:

```
__declspec( dllimport ) int i;
__declspec( dllexport ) int i;    // Warning; inconsistent;
                                // dllexport takes precedence.
```

- En C++, puede inicializar un puntero de datos local declarado o estático globalmente o con la dirección de un objeto de datos declarado con el `dllimport` atributo, que genera un error en C. Además, puede inicializar un puntero de función local estático con la dirección de una función declarada con el `dllimport` atributo. 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:

```
__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++
}
```

Sin embargo, dado que un programa que incluye el `dllexport` atributo en la declaración de un objeto debe proporcionar la definición de ese objeto en algún lugar del programa, puede inicializar un puntero a función estático global o local con la dirección de una `dllexport` función. Del mismo modo, puede inicializar un puntero de datos estático global o local con la dirección de un `dllexport` objeto de datos. Por ejemplo, el código siguiente no genera errores en C ni en 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 `dllexport` a una clase regular que tiene una clase base que no está marcada como `dllexport`, el compilador generará C4275.

El compilador genera la misma advertencia si la clase base es una especialización de una plantilla de clase. Para solucionar este fin, marque la clase base con `dllexport`. El problema con una especialización de una plantilla de clase es dónde colocar el `__declspec(dllexport)`; no se permite marcar la plantilla de clase. En su lugar, cree explícitamente una instancia de la plantilla de clase y marque esta creación de instancias explícita con `dllexport`. Por ejemplo:

```

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:

```

class __declspec(dllexport) D : public B<D> {
// ...

```

Dado que se trata de 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 se aplica implícitamente `dllexport` a las especializaciones de plantillas de clase. Puede hacer lo siguiente y no recibir una advertencia:

```

class __declspec(dllexport) D : public B<D> {
// ...

```

FIN de Específicos de Microsoft

Vea también

[dllexport, dllimport](#)

Utilizar `dllimport` y `dllexport` en las clases de C++

06/03/2021 • 7 minutes to read • [Edit Online](#)

Específicos de Microsoft

Puede declarar clases de C++ con el `dllimport` `dllexport` atributo o. 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:

```
#define DllExport __declspec( dllexport )

class DllExport C {
    int i;
    virtual int func( void ) { return 1; }
};
```

Tenga en cuenta que está prohibido el uso explícito de los `dllimport` `dllexport` atributos y en los miembros de una clase exportable.

Clases `dllexport`

Al declarar una clase `dllexport`, se exportan todas sus funciones miembro y 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)

Al declarar una clase `DllImport`, se importan todas sus funciones miembro y miembros de datos estáticos. A diferencia del comportamiento de `DllImport` y `dllexport` en los tipos que no son de clase, los miembros de datos estáticos no pueden especificar una definición en el mismo programa en el que `DllImport` se define una clase.

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 `dllexport` que una clase herede de una `DllImport` clase y una `DllImport` clase que herede de una `dllexport` clase (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 de miembro selectivo

Dado que las funciones miembro y los datos estáticos dentro de una clase tienen implícitamente una

vinculación externa, puede declararlas con el `dllimport` o `dllexport` atributo o, a menos que se exporte toda la clase. Si se importa o se exporta la clase completa, se prohíbe la declaración explícita de las funciones miembro y los datos como `dllimport` o `dllexport`. Si declara un miembro de datos estático dentro de una definición de clase como `dllexport`, una definición debe aparecer en alguna parte dentro del mismo programa (como con la vinculación externa que no es de clase).

Del mismo modo, puede declarar funciones miembro con `dllimport` los `dllexport` atributos o. En este caso, debe proporcionar una `dllexport` definición 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 `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 `dllimport` o `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 la que la declaró, se genera una advertencia si la función se define como `dllexport` o `dllimport` (si esta definición difiere de la especificada en la declaración de clase).

FIN de Específicos de Microsoft

Vea también

[dllexport, dllimport](#)

jitintrinsic

06/03/2021 • 2 minutes to read • [Edit Online](#)

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)
```

Observaciones

`jitintrinsic` agrega un MODOPT (`IsJitIntrinsic`) a una firma de función.

No se recomienda a los usuarios usar este `__declspec` modificador, ya que pueden producirse resultados inesperados.

Consulta también

[__declspec](#)

[Palabras clave](#)

naked (C++)

06/03/2021 • 3 minutes to read • [Edit Online](#)

Específicos de Microsoft

En el caso de las funciones declaradas con el `naked` atributo, el compilador genera código sin código de prólogo y 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 `naked` atributo solo es válido en x86 y ARM, y no está disponible en x64.

Sintaxis

```
_declspec(naked) declarator
```

Observaciones

Dado `naked` que el atributo solo es pertinente para la definición de una función y no es un modificador de tipo, las funciones Naked deben usar la sintaxis de atributo extendido y la palabra clave `_declspec`.

El compilador no puede generar una función insertada para una función marcada con el atributo Naked, aunque la función también esté marcada con la palabra clave `_forceinline`.

El compilador emite un error si el `naked` atributo se aplica a un valor distinto de la definición de un método no miembro.

Ejemplos

Este código define una función con el `naked` atributo:

```
_declspec( naked ) int func( formal_parameters ) {}
```

O bien, como alternativa:

```
#define Naked _declspec( naked )
Naked int func( formal_parameters ) {}
```

El `naked` atributo 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 lo tanto, el `naked` atributo no se considera parte del tipo de la función y los punteros de función no pueden tener el `naked` atributo. Además, el `naked` atributo 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 `naked` atributo 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

Vea también

[__declspec](#)

[Palabras clave](#)

[Llamadas a funciones Naked](#)

Específico de Microsoft

`noalias` significa que una llamada de función no modifica ni hace referencia al estado global visible y solo modifica la memoria a la que apunta *directamente* por los parámetros de puntero (direcciónamientos de primer nivel).

Si una función se anota como `noalias`, el optimizador puede suponer que solo se hace referencia a los propios parámetros, y solo los direcciónamientos indirectos de primer nivel de los parámetros de puntero, dentro de la función.

La `noalias` anotación solo se aplica dentro del cuerpo de la función anotada. Marcar una función como `_declspec(noalias)` no afecta a los alias de los punteros devueltos por la función.

Para obtener otra anotación que pueda afectar al alias, vea [`_declspec\(restrict\)`](#).

Ejemplo

En el siguiente ejemplo se muestra el uso de `_declspec(noalias)`.

Cuando `multiply` se anota la función que tiene acceso a la memoria `_declspec(noalias)`, se indica al compilador que esta función no modifica el estado global excepto a través de los punteros de su lista de parámetros.

```

// 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);
}

```

Consulta también

`__declspec`

Palabras clave

`__declspec(restrict)`

noinline

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

`__declspec(noinline)` indica al compilador que nunca inlinee 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 se marca una función `noinline`, la función de llamada será más pequeña y, por tanto, una candidata para la inserción del compilador.

```
class X {  
    __declspec(noinline) int mbrfunc() {  
        return 0;  
    } // will not inline  
};
```

FIN de Específicos de Microsoft

Vea también

[__declspec](#)

[Palabras clave](#)

[inline, __inline, __forceinline](#)

noreturn

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Este `__declspec` atributo indica al compilador que una función no devuelve un valor. Como consecuencia, el compilador sabe que `__declspec(noreturn)` no se puede tener acceso al código que sigue a una llamada a una función.

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 del control debido a una función que no devuelve nunca, puede usar `__declspec(noreturn)` para evitar esta advertencia o error.

NOTE

Agregar `__declspec(noreturn)` a una función que se espera que devuelva puede producir un comportamiento indefinido.

Ejemplo

En el ejemplo siguiente, la `else` cláusula no contiene una instrucción `return`. Al declarar `fatal` como se `__declspec(noreturn)` evita un mensaje de error o de advertencia.

```
// 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

Vea también

[__declspec](#)

[Palabras clave](#)

no_sanitize_address

09/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

El `__declspec(no_sanitize_address)` especificador indica al compilador que deshabilite el saneado de direcciones en funciones, variables locales o variables globales. Este especificador se usa junto con [AddressSanitizer](#).

NOTE

`__declspec(no_sanitize_address)` deshabilita el comportamiento *del compilador*, no el comportamiento *en tiempo de ejecución*.

Ejemplo

Vea la [referencia de compilación de AddressSanitizer](#) para obtener ejemplos.

FIN de Específicos de Microsoft

Vea también

[__declspec](#)

[Palabra](#)

[AddressSanitizer](#)

nothrow (C++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

`__declspec` Atributo extendido que se puede utilizar en la declaración de funciones.

Sintaxis

`__declspec de tipo de valor devuelto (nothrow) [Convención de llamada] nombre de función ([lista de argumentos])`

Observaciones

Se recomienda que todo el código nuevo use el operador `noexception` 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 se invoque `STD::Terminate`, a diferencia de `noexcept` o en el 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 siguiente directiva de preprocesador, las tres declaraciones de función siguientes son equivalentes en el modo `/STD: c++ 14`:

```
#define WINAPI __declspec(nothrow) __stdcall  
  
void WINAPI f1();  
void __declspec(nothrow) __stdcall f2();  
void __stdcall f3() throw();
```

En `/STD: modo 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 utiliza la sintaxis definida por el estándar de C++. En C++ 17, la `throw()` palabra clave quedó en desuso.

FIN de Específicos de Microsoft

Vea también

[__declspec](#)
[noexcept](#)
[Palabras clave](#)

novtable

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Este es un `__declspec` atributo extendido.

Esta forma de `__declspec` se puede aplicar a cualquier declaración de clase, pero solo debe aplicarse a las clases de interfaz puras, es decir, las clases de las que nunca se crearán instancias por sí mismas. `__declspec` Impide que el compilador genere código para inicializar vptr en los constructores y en 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, tiene acceso a un miembro de clase, recibirá una infracción de acceso (AV).

Ejemplo

```
// 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();
}
```

In Y

FIN de Específicos de Microsoft

Vea también

[__declspec](#)

[Palabras clave](#)

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 se diseñó principalmente para usarse al compilar con `/clr:pure`, que está en desuso en visual studio 2015 y no se admite en visual studio 2017. Al compilar con `/clr`, las variables globales y estáticas son por proceso de forma predeterminada y no necesitan usar `__declspec(process)`.

Solo se puede marcar con una variable global, una variable miembro estática o una variable local estática de tipo nativo `__declspec(process)`.

`process` solo es válido cuando se compila con `/clr`.

Si desea que cada dominio de aplicación tenga su propia copia de una variable global, utilice [AppDomain](#).

Vea [dominios de aplicación y Visual C++](#) para obtener más información.

Consulta también

[__declspec](#)

[Palabras clave](#)

propiedad (C++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

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
```

Observaciones

Cuando el compilador ve un miembro de datos declarado con este atributo a la derecha de un operador de selección de miembro ("." o "->"), convierte la operación en una `get` o `put` función o, en función de si dicha expresión es un valor l o un valor r. En contextos más complicados, como "`+=`", se realiza una reescritura mediante el `get` y el `put`.

Este atributo se puede utilizar también en la declaración de una matriz vacía en una definición de clase o estructura. Por ejemplo:

```
__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

```
// 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

06/03/2021 • 3 minutes to read • [Edit Online](#)

Específicos de Microsoft

Cuando se aplica a una declaración o definición de función que devuelve un tipo de puntero, `restrict` indica al compilador que la función devuelve un objeto que no tiene *alias*, es decir, que hace referencia a otro puntero. Esto permite al compilador realizar optimizaciones adicionales.

Sintaxis

```
* __declspec(restrict) ***pointer_return_type función();
```

Observaciones

El compilador propaga `__declspec(restrict)`. Por ejemplo, la función de CRT `malloc` tiene una `__declspec(restrict)` decoración y, por consiguiente, el compilador supone que los punteros inicializados en las ubicaciones de memoria de `malloc` tampoco están alias por punteros previamente existentes.

El compilador no comprueba que el puntero devuelto no tiene un alias en realidad. Es responsabilidad del desarrollador asegurarse de que el programa no alias de un puntero marcado con el modificador `restrict __declspec`.

Para obtener una semántica similar en las variables, vea [__restrict](#).

Para otra anotación que se aplique a los alias en una función, vea [__declspec \(noalias\)](#).

Para obtener información sobre la `restrict` palabra clave que forma parte de C++ &, vea [restrict \(C++ &\)](#).

Ejemplo

En el siguiente ejemplo se muestra el uso de `__declspec(restrict)`.

Cuando `__declspec(restrict)` se aplica a una función que devuelve un puntero, 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. Sin embargo, se utilizan en `ma` y su llamador `init` de forma que devuelve memoria a la que no hace referencia el programa, 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 se puede incluir en el alias de los punteros existentes.

```

// 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);
}

```

Vea también

[Palabras clave](#)

[__declspec](#)

[__declspec \(noalias\)](#)

safebuffers

06/03/2021 • 3 minutes to read • [Edit Online](#)

Específicos de Microsoft

Indica al compilador que no inserte comprobaciones de seguridad de saturación del búfer para una función.

Sintaxis

```
__declspec( safebuffers )
```

Observaciones

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 válidos 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, vea [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, puede suprimir las comprobaciones de seguridad de una función aplicando la `__declspec(safebuffers)` palabra clave a la declaración de función.

Caution

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 usar una [palabra clave](#) de inserción para insertar una copia de una *función secundaria*. Si la `__declspec(safebuffers)` palabra clave se aplica a una función, se suprime la detección de saturación del búfer para esa función. Sin embargo, la inclusión afecta a la `__declspec(safebuffers)` palabra clave de las siguientes maneras.

Supongamos que la opción del compilador **/GS** se especifica para ambas funciones, pero la función principal especifica la `__declspec(safebuffers)` palabra clave. 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 obligar al compilador a alinear esa función independientemente de las optimizaciones del compilador.
- Dado que la función secundaria es válida para las comprobaciones de seguridad, las comprobaciones de seguridad también se aplican a la función principal aunque especifique la `__declspec(safebuffers)` palabra clave.

Ejemplo

En el código siguiente se muestra cómo usar la `__declspec(safebuffers)` palabra clave.

```
// 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;
}
```

FIN de Específicos de Microsoft

Vea también

[__declspec](#)

[Palabras clave](#)

[inline, __inline, __forceinline](#)

[strict_gs_check](#)

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
```

Observaciones

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 vinculador `/OPT:REF` (optimizaciones), se producirá la eliminación de COMDAT para quitar todos los elementos de datos sin referencia en la salida del vinculador.

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.

En el caso de los objetos globales inicializados dinámicamente, `selectany` descartará también el código de inicialización de un objeto sin referencia.

Un elemento de datos globales se puede inicializar normalmente solo una vez en un proyecto EXE o DLL.

`selectany` se puede usar para inicializar los datos globales definidos por encabezados cuando el mismo encabezado aparece en más de un archivo de código fuente. `selectany` está disponible en los compiladores de C y C++.

NOTE

`selectany` solo se puede aplicar a la inicialización real de elementos de datos globales que son visibles externamente.

Ejemplo: `selectany` atributo

Este código muestra cómo usar el `selectany` atributo:

```

//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: usar el `selectany` atributo para garantizar el plegamiento de COMDAT de datos

Este código muestra cómo usar el `selectany` atributo para garantizar el plegamiento de COMDAT de datos cuando también se usa la `/OPT:ICF` opción del vinculador. Tenga en cuenta que los datos se deben marcar con `selectany` y colocar en una `const` sección (de solo lectura). Debe especificar explícitamente la sección de solo lectura.

```

// 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

Vea también

[__declspec](#)

[Palabras clave](#)

spectre

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Indica al compilador que no inserte instrucciones de barrera de ejecución especulativa de Spectre Variant 1 para una función.

Sintaxis

```
__declspec (Spectre (nomitigación))
```

Observaciones

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 Spectre Variant 1. 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 requiere un rendimiento máximo.

El análisis experto podría determinar que una función es segura desde un defecto de la comprobación de límites de Spectre Variant 1. En ese caso, puede suprimir la generación de código de mitigación dentro de una función aplicando `__declspec(spectre(nomitigation))` a la declaración de función.

Caution

Las instrucciones de barrera de ejecución especulativa de [/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))`:

```
// 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

Vea también

[__declspec](#)

Palabras clave
[/Qspectre](#)

subproceso

06/03/2021 • 6 minutes to read • [Edit Online](#)

Específicos de Microsoft

El `thread` modificador de clase de almacenamiento extendido se usa para declarar una variable local de subproceso. Para el equivalente portable en C++ 11 y versiones posteriores, use el especificador de clase de almacenamiento `thread_local` para el código portable. En Windows `thread_local` se implementa con `__declspec(thread)`.

Sintaxis

* `__declspec(thread) ***declarador`

Observaciones

El almacenamiento local para el subproceso (TLS) es el mecanismo por el que cada subproceso de un proceso con varios subproceso asigna almacenamiento para los datos específicos de ese subproceso. En los programas multiproceso estándar, los datos se comparten entre todos los subproceso de un proceso dado, mientras que el almacenamiento local para el subproceso es el mecanismo para asignar datos por subproceso. Para obtener una descripción completa de los subprocesos, vea [multithreading](#).

Las declaraciones de variables locales de subproceso deben utilizar la [Sintaxis de atributo extendido](#) y la `__declspec` palabra clave con la `thread` palabra clave. Por ejemplo, el código siguiente declara una variable local de subproceso de entero y la inicializa con un valor:

```
__declspec( thread ) int tls_i = 1;
```

Al usar variables locales de subproceso en bibliotecas cargadas dinámicamente, debe tener en cuenta los factores que pueden hacer que una variable local de subproceso 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 subproceso que provocó la carga del archivo binario o DLL en el proceso, y para los subprocesos que se iniciaron después de cargar el archivo binario o DLL. No se llama a las funciones de inicialización para ningún otro subproceso 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 recibe ese mensaje si la DLL no está en el proceso cuando se inicia el subproceso.
2. Las variables locales de subproceso que se inicializan estáticamente con valores constantes normalmente se inicializan correctamente en todos los subprocesos. Sin embargo, a partir del 2017 de diciembre hay un problema de conformidad conocido en el compilador de Microsoft C++, en el que `constexpr` las variables reciben Dynamic en lugar de una inicialización estática.

Nota: se espera que ambos problemas se solucionen en futuras actualizaciones del compilador.

Además, debe seguir estas instrucciones al declarar objetos y variables locales para el subproceso:

- Solo puede aplicar el `thread` atributo a las declaraciones y definiciones de clase y datos; `thread` no se puede usar en declaraciones o definiciones de función.
- Solo puede especificar el `thread` atributo en elementos de datos con duración de almacenamiento

estática. Esto incluye los objetos de datos globales (`static` y `extern`), los objetos estáticos locales y los miembros de datos estáticos de las clases. No se pueden declarar objetos de datos automáticos con el `thread` atributo.

- Debe utilizar el `thread` atributo para la declaración y la definición de un objeto local de subprocesso, tanto si la declaración y la definición se producen en el mismo archivo como en archivos independientes.
- No se puede usar el `thread` atributo como un modificador de tipo.
- Dado que se permite la declaración de objetos que usan el `thread` atributo, estos dos ejemplos son semánticamente equivalentes:

```
// 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.
```

- Estándar C permite la inicialización de un objeto o una variable con una expresión que contenga una referencia a sí misma, pero solo para objetos no estáticos. Aunque C++ normalmente permite la inicialización dinámica de un objeto con una expresión que contenga una referencia a sí misma, este tipo de inicialización no se permite con objetos locales de subprocesso. Por ejemplo:

```
// 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 `sizeof` expresión que incluye el objeto que se va a inicializar no constituye una referencia a sí misma y se permite en C y C++.

FIN de Específicos de Microsoft

Vea también

[__declspec](#)

[Palabras clave](#)

[Almacenamiento local para el subprocesso \(TLS\)](#)

uuid (C++)

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

El compilador asocia un GUID a una clase o estructura declarada o definida (solo en definiciones de objetos COM completos) con el `uuid` atributo.

Sintaxis

```
__declspec( uuid("ComObjectGUID") ) declarator
```

Observaciones

El `uuid` atributo toma una cadena como argumento. Esta cadena denomina un GUID en formato de registro normal con o sin los delimitadores {} . Por ejemplo:

```
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 del sistema suministren las definiciones de interfaces como `IUnknown` y la nueva declaración en otro encabezado (como `<comdef.h>`) para proporcionar 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

Vea también

[__declspec](#)

[Palabras clave](#)

__restrict

06/03/2021 • 2 minutes to read • [Edit Online](#)

Al igual que el modificador `__declspec` (`restrict`) , la `__restrict` palabra clave (dos subrayados iniciales ' _ ') indica que un símbolo no tiene un alias en el ámbito actual. La `__restrict` palabra clave difiere del `__declspec (restrict)` modificador de las maneras siguientes:

- La `__restrict` palabra clave solo es válida en variables y `__declspec (restrict)` solo es válida en declaraciones y definiciones de función.
- `__restrict` es similar a `restrict` para C a partir de C99, pero `__restrict` se puede usar en programas de C++ y C.
- Cuando `__restrict` se usa, el compilador no propagará la propiedad no alias de una variable. Es decir, si se asigna una `__restrict` variable a una variable que no es `__restrict` , el compilador seguirá permitiendo el alias de la variable que no es de `__restrict`. Esto es diferente del comportamiento de la palabra clave del lenguaje C99 C `restrict` .

Por lo general, si desea influir en el comportamiento de una función completa, use `__declspec (restrict)` en lugar de la palabra clave.

Por 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, `__restrict` se puede usar en las referencias de C++.

NOTE

Cuando se usa en una variable que también tiene la `volatile` palabra clave, `volatile` tendrá prioridad.

Ejemplo

```
// __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;
};
```

Vea también

[Palabras clave](#)

`_sptr, __uptr`

06/03/2021 • 5 minutes to read • [Edit Online](#)

Específicos de Microsoft

Utilice el `_sptr` `__uptr` modificador o en una declaración de puntero de 32 bits para especificar cómo el compilador convierte 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 0x000000007FFFFFFF, pero la dirección de 32 bits 0x80000000 cambia incorrectamente a 0xFFFFFFFF80000000.

El `_sptr` modificador o puntero con signo especifica que una conversión de puntero establece los bits más significativos de un puntero de 64 bits en el bit de signo del puntero de 32 bits. El `__uptr` modificador o el puntero sin signo especifica que una conversión establezca los bits más significativos en cero. Las declaraciones siguientes muestran los `_sptr` `__uptr` modificadores y utilizados con dos punteros incompletos, dos punteros calificados con el tipo de `_ptr32` y un parámetro de función.

```
int * __sptr psp;
int * __uptr pup;
int * __ptr32 __sptr psp32;
int * __ptr32 __uptr pup32;
void MyFunction(char * __uptr __ptr32 myValue);
```

Use los `_sptr` `__uptr` modificadores y 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 pueden 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 para `_sptr` y `__uptr` a menos que se especifique la opción del compilador [/za \(deshabilitar extensiones de lenguaje\)](#).

Ejemplo

En el ejemplo siguiente se declaran punteros de 32 bits que usan los `_sptr` `__uptr` modificadores y, se asigna cada puntero de 32 bits a una variable de puntero de 64 bits y, a continuación, se muestra el valor hexadecimal de cada puntero de bit de 64. El ejemplo se compila con el compilador de 64 bits nativo y se ejecuta en una plataforma de 64 bits.

```

// 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;
}

```

```

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

Vea también

[Modificadores específicos de Microsoft](#)

`_unaligned`

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específico de Microsoft. Cuando se declara un puntero con el `_unaligned` modificador, el compilador supone que el puntero direcciona datos que no están alineados. Por consiguiente, se genera código adecuado para la plataforma para controlar las lecturas y escrituras sin alinear a través del puntero.

Observaciones

Este modificador describe la alineación de los datos direccionados por el puntero; se supone que el propio puntero está alineado.

La necesidad de la `_unaligned` palabra clave varía según la plataforma y el entorno. Si no se marcan correctamente los datos, pueden producirse problemas que van desde penalizaciones de rendimiento a errores de hardware. El `_unaligned` modificador no es válido para la plataforma x86.

Por 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](#)
- [alignof Operator](#)
- [pack](#)
- [/zp](#) ([Alineación de miembros de estructura](#))
- [Ejemplos de alineación de estructuras](#)

Consulta también

[Palabras clave](#)

`_w64`

06/03/2021 • 3 minutes to read • [Edit Online](#)

Esta palabra clave específica de Microsoft está obsoleta. En las versiones de Visual Studio anteriores a Visual Studio 2013, esto permite marcar variables, de modo que, al compilar con `/Wp64`, el compilador informará de las advertencias que se notificarían si se compilara con un compilador de 64 bits.

Sintaxis

tipo `_w64` de *identificador* de

Parámetros

type

Uno de los tres tipos que pueden causar problemas en el código que se traslada de un compilador de 32 bits a un compilador de 64 bits: `int`, `long` o un puntero.

identifier

Identificador de la variable que va a crear.

Observaciones

IMPORTANT

La opción del compilador `/Wp64` y la `_w64` palabra clave están en desuso en Visual Studio 2010 y Visual Studio 2013 y se han quitado a partir de Visual Studio 2013. Si usa la `/Wp64` opción del compilador en la línea de comandos, el compilador emite Command-Line ADVERTENCIA D9002. La `_w64` palabra clave se pasa por alto de forma silenciosa. En lugar de utilizar esta opción y la palabra clave para detectar problemas de portabilidad de 64 bits, use un compilador de Microsoft C++ que tenga como destino 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 tenga `_w64` en ella debe ser de 32 bits en x86 y 64 bits en x64.

Para detectar problemas de portabilidad con las versiones del compilador de Microsoft C++ anteriores a Visual Studio 2010, `_w64` se debe especificar la palabra clave en cualquier `typedefs` que cambie el tamaño entre las plataformas de 32 bits y de bits de 64. Para cualquier tipo de este tipo, `_w64` debe aparecer solo en la definición de 32 bits de `TypeDef`.

Por compatibilidad con versiones anteriores, `_w64` es un sinónimo de `_w64` a menos que se especifique la opción del compilador `/za` ([Disable Language Extensions](#)) .

La `_w64` palabra clave se omite si la compilación no utiliza `/Wp64`.

Para obtener más información sobre la portabilidad a 64 bits, vea los temas siguientes:

- [Opciones del compilador de MSVC](#)
- [Trasladar código de 32 bits a código de 64 bits](#)
- [Configuración de Visual C++ en destinos de 64 bits, x64](#)

Ejemplo

```
// __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

06/03/2021 • 2 minutes to read • [Edit Online](#)

(C++ 11) El identificador predefinido `__FUNC__` se define implícitamente como una cadena que contiene el nombre no completo y sin adornar de la función de inclusión. `__FUNC__` está asignada por el estándar de C++ y no es una extensión de Microsoft.

Sintaxis

```
__func__
```

Valor devuelto

Devuelve una matriz const char terminada en NULL de caracteres que contiene el nombre de la función.

Ejemplo

```
#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

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

El compilador de Microsoft C++ puede leer directamente bibliotecas de tipos de modelo de objetos componentes (COM) y traducir el contenido en código fuente de C++ que se puede incluir en la compilación. Las extensiones de lenguaje están disponibles para facilitar la programación COM en el lado cliente de las aplicaciones de escritorio.

Mediante el uso de la [Directiva de preprocesador de #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 `__declspec` atributo extendido para asignar un identificador único global (GUID) a un objeto com. La palabra clave `__uuidof` se puede utilizar para extraer el GUID asociado a un objeto com. Otro `__declspec` atributo, [propiedad](#), se puede utilizar para especificar los `get` `set` métodos y para un miembro de datos de un objeto com.

Se proporciona un conjunto de funciones y clases globales que admiten COM para admitir los `VARIANT` `BSTR` tipos y, implementar punteros inteligentes y encapsular el objeto de error producido por `_com_raise_error` :

- [Funciones globales COM del compilador](#)
- [_bstr_t](#)
- [_com_error](#)
- [_com_ptr_t](#)
- [_variant_t](#)

FIN de Específicos de Microsoft

Vea también

[Clases de compatibilidad con COM del compilador](#)

[Funciones globales COM del compilador](#)

Funciones globales COM de compilador

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Están disponibles las siguientes rutinas:

FUNCIÓN	DESCRIPCIÓN
_com_raise_error	Produce una _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

Vea también

[Clases de compatibilidad con COM del compilador](#)

[Compatibilidad con COM del compilador](#)

_com_raise_error

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Produce una [_com_error](#) en respuesta a un error.

Sintaxis

```
void __stdcall _com_raise_error(
    HRESULT hr,
    IErrorInfo* perrinfo = 0
);
```

Parámetros

h

Información de HRESULT.

perrinfo

Objeto [IErrorInfo](#).

Observaciones

`_com_raise_error`, que se define en `<comdef.h>`, se puede reemplazar por una versión escrita por el usuario del 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 `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 puede utilizar [_set_com_error_handler](#) para reemplazar la función de control de errores predeterminada.

De forma predeterminada, `_com_raise_error` se define de la siguiente manera:

```
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 el **wchar_t es Native Type** (opción del compilador) es on, use omsuppw.lib o comsuppwd.lib. Si **wchar_t es el tipo nativo** está desactivado, use comsupp.lib. Para obtener más información, vea [/Zc: wchar_t \(Wchar_t es tipo nativo\)](#).

Consulta también

[Funciones globales COM del compilador](#)

_set_com_error_handler

ConvertStringToBSTR

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Convierte un valor `char *` en un objeto `BSTR`.

Sintaxis

```
BSTR __stdcall ConvertStringToBSTR(const char* pSrc)
```

Parámetros

pSrc

Una `char *` variable.

Ejemplo

```
// 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);
}
```

```
char * text: Test
BSTR text: Test
```

FIN de Específicos de Microsoft

Requisitos

Encabezado: <comutil.h>

Lib: omsuppw.lib o comsuppwd.lib (vea [/Zc: Wchar_t \(Wchar_t es tipo nativo\)](#) para obtener más información)

Consulta también

[Funciones globales COM del compilador](#)

ConvertBSTRToString

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Convierte un valor `BSTR` en un objeto `char *`.

Sintaxis

```
char* __stdcall ConvertBSTRToString(BSTR pSrc);
```

Parámetros

pSrc

Una variable BSTR.

Observaciones

`ConvertBSTRToString` asigna una cadena que se debe eliminar.

Ejemplo

```
// 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;
}
```

```
BSTR text: Test
char * text: Test
```

FIN de Específicos de Microsoft

Requisitos

Encabezado: <comutil.h>

Lib: omsuppw.lib o comsuppwd.lib (vea [/Zc: Wchar_t \(Wchar_t es tipo nativo\)](#) para obtener más información)

Consulta también

[Funciones globales COM del compilador](#)

_set_com_error_handler

06/03/2021 • 2 minutes to read • [Edit Online](#)

Reemplaza la función predeterminada que se utiliza para el control de errores de COM.
`_set_com_error_handler` es específico de Microsoft.

Sintaxis

```
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`.

Observaciones

De forma predeterminada, `_com_raise_error` controla todos los errores de com. Puede cambiar este comportamiento mediante el uso de `_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

```

// _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;
}

```

Exception raised: Unable to establish the connection!

Requisitos

Encabezado: <comdef.h>

Lib: Si se especifica la opción del compilador /Zc: wchar_t (valor predeterminado), use omsuppw.lib o comsuppwd.lib. Si se especifica la opción de compilador /Zc: wchar_t, use comsupp.lib. Para obtener más información, incluida la forma de establecer esta opción en el IDE, vea [/Zc: wchar_t \(wchar_t es tipo nativo\)](#).

Consulta también

[Funciones globales COM del compilador](#)

Clases de compatibilidad con COM del compilador

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Las clases estándar se utilizan para proporcionar compatibilidad con algunos de los tipos COM. Las clases se definen en `<comdef.h>` y los archivos de encabezado generados a partir de 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 los 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

Vea también

[Compatibilidad con COM del compilador](#)

[Funciones globales COM del compilador](#)

[Referencia del lenguaje C++](#)

Clase `_bstr_t`

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Un `_bstr_t` objeto encapsula el [tipo de datos BSTR](#). La clase administra la asignación y desasignación de recursos a través de llamadas a funciones a `SysAllocString` y `SysFreeString` y otras `BSTR` API cuando sea necesario. La `_bstr_t` clase 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

OPERATOR	DESCRIPCIÓN
<code>operator =</code>	Asigna un nuevo valor a un objeto <code>_bstr_t</code> existente.
<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 encapsulado <code>BSTR</code> es una cadena nula.

OPERATOR	DESCRIPCIÓN
<pre>operator == operator != operator < operator > operator <= operator >=</pre>	Compara dos objetos <code>_bstr_t</code> .
<pre>operator wchar_t* operator char*</pre>	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, vea [/Zc:wchar_t](#) (`wchar_t` es un tipo nativo))

Vea también

[Clases de compatibilidad con COM del compilador](#)

`_bstr_t::_bstr_t`

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Construye un objeto `_bstr_t`.

Sintaxis

```
_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 `false` es, el `bstr` argumento se adjunta al nuevo objeto sin realizar una copia mediante una llamada a `SysAllocString`.

Comentarios

La `_bstr_t` clase proporciona varios constructores:

`_bstr_t()`

Construye un objeto predeterminado `_bstr_t` que encapsula un `BSTR` objeto 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 encapsulado `BSTR` 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 `_bstr_t` objeto a partir de un `_variant_t` objeto recuperando primero un `BSTR` objeto del objeto encapsulado `VARIANT`.

```
_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 proporcionado `BSTR` se adjunta al nuevo objeto sin crear una nueva copia mediante `SysAllocString`. Las funciones de contenedor de los encabezados de la biblioteca de tipos usan este constructor para encapsular y tomar la propiedad de un `BSTR` devuelto por un método de interfaz.

FIN de Específicos de Microsoft

Vea también

`_bstr_t` las

`_variant_t` las

`bstr_t::Assign`

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Copia un valor `BSTR` en el valor `BSTR` contenido en `_bstr_t`.

Sintaxis

```
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 la longitud completa de `BSTR`, sea cual sea el contenido.

Ejemplo

```

// _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);

    // resuse 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));
}

```

```
bstrWrapper = some text
bstrWrapper = (null)
bstr = some text
bstrWrapper = SysAllocedString
bstrWrapper = some text
bstrWrapper = Yet another string
bstrWrapper2 = some text
bstrWrapper = changing BSTR
bstrWrapper2 = some text
```

FIN de Específicos de Microsoft

Vea también

[_bstr_t](#) las

`_bstr_t::Attach`

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Vincula un contenedor `_bstr_t` a un `BSTR`.

Sintaxis

```
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

Vea [`_bstr_t::Assign`](#) para obtener un ejemplo de uso de `Attach`.

FIN de Específicos de Microsoft

Vea también

[`_bstr_t`](#) Las

`_bstr_t::copy`

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Crea una copia del objeto `BSTR` encapsulado.

Sintaxis

```
BSTR copy( bool fCopy = true ) const;
```

Parámetros

`fCopy`
Si `true` `copy` es, devuelve una copia de la contenida `BSTR`; de lo contrario, `copy` devuelve el actual `BSTR`.

Comentarios

Devuelve una copia recién asignada del objeto encapsulado `BSTR` o el propio objeto encapsulado, en función del parámetro.

Ejemplo

```
STDMETHODIMP CAlertMsg::get_ConnectionStr(BSTR *pVal){ // m_bsConStr is _bstr_t
    *pVal = m_bsConStr.copy();
}
```

FIN de Específicos de Microsoft

Vea también

[_bstr_t](#) las

`_bstr_t::Detach`

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Devuelve el `BSTR` contenido en `_bstr_t` y desasocia `BSTR` de `_bstr_t`.

Sintaxis

```
BSTR Detach( ) throw;
```

Valor devuelto

Devuelve el `BSTR` encapsulado por `_bstr_t`.

Ejemplo

Vea [`_bstr_t::Assign`](#) para obtener un ejemplo que usa `Detach`.

FIN de Específicos de Microsoft

Vea también

[`_bstr_t`](#) las

`_bstr_t::GetAddress`

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Libera cualquier cadena existente y devuelve la dirección de una cadena recién asignada.

Sintaxis

```
BSTR* GetAddress( );
```

Valor devuelto

Puntero a `BSTR` contenido en `_bstr_t`.

Comentarios

`GetAddress` afecta a todos los `_bstr_t` objetos que comparten un `BSTR`. Más de uno `_bstr_t` puede compartir un `BSTR` mediante el uso del constructor de copias y `operator=`.

Ejemplo

Vea [`_bstr_t::Assign`](#) para obtener un ejemplo que usa `GetAddress`.

FIN de Específicos de Microsoft

Vea también

[`_bstr_t`](#) las

`_bstr_t::GetBSTR`

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Señala al principio del objeto `BSTR` incluido en `_bstr_t`.

Sintaxis

```
BSTR& GetBSTR( );
```

Valor devuelto

Principio del objeto `BSTR` incluido en `_bstr_t`.

Comentarios

`GetBSTR` afecta a todos los `_bstr_t` objetos que comparten un `BSTR`. Más de uno `_bstr_t` puede compartir un `BSTR` mediante el uso del constructor de copias y `operator=`.

Ejemplo

Vea [`_bstr_t::Assign`](#) para obtener un ejemplo que usa `GetBSTR`.

FIN de Específicos de Microsoft

Vea también

[`_bstr_t`](#) las

`_bstr_t::length`

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Devuelve el número de caracteres de `_bstr_t`, sin incluir el carácter null de terminación, del `BSTR` encapsulado.

Sintaxis

```
unsigned int length( ) const throw( );
```

Comentarios

FIN de Específicos de Microsoft

Vea también

`_bstr_t` las

`_bstr_t::operator =`

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Asigna un nuevo valor a un objeto `_bstr_t` existente.

Sintaxis

```
_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

Vea [`_bstr_t::Assign`](#) para obtener un ejemplo que usa `operator=`.

Vea también

[`_bstr_t`](#) las

`_bstr_t::operator +=`, `_bstr_t::operator +`

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Anexa caracteres al final del `_bstr_t` objeto o concatena dos cadenas.

Sintaxis

```
_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 del encapsulado `BSTR` de `s1` al final del encapsulado de este objeto `BSTR`.
- `operator+(s1)` Devuelve el nuevo `_bstr_t` objeto que se forma mediante la concatenación de este objeto `BSTR` y el de la estructura `s1`.
- `operator+(s2, s1)` Devuelve un nuevo `_bstr_t` que se forma mediante la concatenación de una cadena multibyte `s2`, convertida en Unicode y la `BSTR` encapsulada en `s1`.
- `operator+(s3, s1)` Devuelve un nuevo `_bstr_t` que se forma mediante la concatenación de una cadena Unicode `s3` y la `BSTR` encapsulada en `s1`.

FIN de Específicos de Microsoft

Vea también

`_bstr_t` las

`_bstr_t::operator !`

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Comprueba si el encapsulado `BSTR` es una cadena nula.

Sintaxis

```
bool operator!( ) const throw( );
```

Valor devuelto

Devuelve `true` si el encapsulado `BSTR` es una cadena nula, en `false` caso contrario.

FIN de Específicos de Microsoft

Vea también

[`_bstr_t`](#) las

`_bstr_t` operadores relacionales

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Compara dos objetos `_bstr_t`.

Sintaxis

```
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 contienen; de lo contrario, devuelve `false`.

FIN de Específicos de Microsoft

Vea también

[`_bstr_t` las](#)

`_bstr_t::wchar_t*`, `_bstr_t::char*`

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Devuelve los `BSTR` caracteres como una matriz de caracteres estrechos o anchos.

Sintaxis

```
operator const wchar_t*( ) const throw( );
operator wchar_t*( ) const throw( );
operator const char*( ) const;
operator char*( ) const;
```

Comentarios

Estos operadores se pueden usar para extraer los datos de caracteres encapsulados por el `BSTR` objeto. Al asignar un nuevo valor al puntero devuelto no se modifican los `BSTR` datos originales.

FIN de Específicos de Microsoft

Vea también

`_bstr_t` las

_com_error (Clase)

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Un objeto `_com_error` 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 mediante una de las clases de compatibilidad con com. La clase `_com_error` encapsula el código de error HRESULT y cualquier `IErrorInfo Interface` objeto asociado.

Construcción

NOMBRE	DESCRIPCIÓN
<code>_com_error</code>	Construye un objeto de <code>_com_error</code> .

Operadores

NOMBRE	DESCRIPCIÓN
<code>operador =</code>	Asigna un objeto de <code>_com_error</code> existente a otro.

Funciones de extractor

NOMBRE	DESCRIPCIÓN
<code>Error</code>	Recupera el HRESULT que se pasa 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 HRESULT encapsulado.

Funciones de IErrorInfo

NOMBRE	DESCRIPCIÓN
<code>Descripción</code>	Llama a la función <code>IErrorInfo::GetDescription</code> .
<code>HelpContext</code>	Llama a la función <code>IErrorInfo::GetHelpContext</code> .
<code>HelpFile</code>	Llama a la función <code>IErrorInfo::GetHelpFile</code> .
<code>Origen</code>	Llama a la función <code>IErrorInfo::GetSource</code> .
<code>GUID</code>	Llama a la función <code>IErrorInfo::GetGUID</code> .

Extractor de mensajes de formato

NOMBRE	DESCRIPCIÓN
ErrorMessage	Recupera el mensaje de cadena para HRESULT almacenado en el objeto <code>_com_error</code> .

Asignadores de `ExepInfo.wCode` a HRESULT

NOMBRE	DESCRIPCIÓN
<code>HRESULTToWCode</code>	Asigna HRESULT de 32 bits a 16 bits <code>wCode</code> .
<code>WCodeToHRESULT</code>	Asigna de 16 bits <code>wCode</code> a HRESULT de 32 bits.

FIN de Específicos de Microsoft

Requisitos

Encabezado:<comdef.h>

Lib: omsuppw.lib o comsuppwd.lib (vea [/Zc:wchar_t \(Wchar_t es tipo nativo\)](#) para obtener más información)

Consulta también

[Clases de compatibilidad con COM del compilador](#)

[IErrorInfo \(interfaz\)](#)

`_com_error` (Funciones miembro)

06/03/2021 • 2 minutes to read • [Edit Online](#)

Para obtener información sobre las funciones miembro de `_com_error`, vea [_com_error \(clase\)](#).

Consulta también

[_com_error \(clase\)](#)

_com_error::_com_error

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Construye un objeto de `_com_error`.

Sintaxis

```
_com_error(
    HRESULT hr,
    IErrorInfo* perrinfo = NULL,
    bool fAddRef=false) throw( );

_com_error( const _com_error& that ) throw( );
```

Parámetros

h

Información de HRESULT.

perrinfo

Objeto `IErrorInfo`.

fAddRef

El valor predeterminado hace que el constructor llame a AddRef en una interfaz que no sea NULL `IErrorInfo`.

Esto proporciona un recuento de referencias correcto en el caso común en el que la propiedad de la interfaz se pasa al objeto `_com_error`, como:

```
throw _com_error(hr, perrinfo);
```

Si no desea que el código transfiera la propiedad al objeto `_com_error`, y `AddRef` es necesario para desplazar `Release` en el destructor `_com_error`, construya el objeto de la manera siguiente:

```
_com_error err(hr, perrinfo, true);
```

that

Objeto de `_com_error` existente.

Observaciones

El primer constructor crea un nuevo objeto a partir de un valor HRESULT y un `IErrorInfo` objeto opcional. La segunda crea una copia de un objeto de `_com_error` existente.

FIN de Específicos de Microsoft

Vea también

[_com_error \(clase\)](#)

_com_error::Description

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Llama a la función `IErrorInfo::GetDescription`.

Sintaxis

```
_bstr_t Description( ) const;
```

Valor devuelto

Devuelve el resultado de `IErrorInfo::GetDescription` para el `IErrorInfo` objeto registrado en el `_com_error` objeto. El `BSTR` resultante se encapsula en un objeto `_bstr_t`. Si no `IErrorInfo` se registra ningún, devuelve un vacío `_bstr_t`.

Observaciones

Llama a la `IErrorInfo::GetDescription` función y recupera los `IErrorInfo` objetos registrados en el `_com_error` objeto. Se omite cualquier error que se produzca mientras se llama al `IErrorInfo::GetDescription` método.

FIN de Específicos de Microsoft

Vea también

[_com_error \(clase\)](#)

_com_error::Error

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Recupera el HRESULT que se pasa al constructor.

Sintaxis

```
HRESULT Error( ) const throw( );
```

Valor devuelto

Elemento HRESULT sin formato pasado en el constructor.

Observaciones

Recupera el elemento HRESULT encapsulado en un `_com_error` objeto.

FIN de Específicos de Microsoft

Vea también

[_com_error \(clase\)](#)

_com_error::ErrorInfo

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Recupera el objeto `IErrorInfo` pasado al constructor.

Sintaxis

```
IErrorInfo * ErrorInfo( ) const throw( );
```

Valor devuelto

Elemento `IErrorInfo` sin formato pasado en el constructor.

Observaciones

Recupera el elemento encapsulado `IErrorInfo` en un `_com_error` objeto o null si no `IErrorInfo` se registra ningún elemento. El llamador debe llamar a `Release` en el objeto devuelto cuando termine de usarlo.

FIN de Específicos de Microsoft

Vea también

[_com_error \(clase\)](#)

_com_error::ErrorMessage

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Recupera el mensaje de cadena para HRESULT almacenado en el objeto `_com_error`.

Sintaxis

```
const TCHAR * ErrorMessage( ) const throw( );
```

Valor devuelto

Devuelve el mensaje de cadena para el HRESULT registrado en el `_com_error` objeto. Si el valor HRESULT es un `wCode` de 16 bits asignado, 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, dependiendo del estado de la macro `_UNICODE`.

Observaciones

Recupera el texto del mensaje del sistema apropiado para HRESULT registrado en el `_com_error` objeto. El texto del mensaje del sistema se obtiene mediante una llamada a la función `FormatMessage` de Win32. La API `FormatMessage` asigna la cadena devuelta, que se libera cuando se destruye el objeto `_com_error`.

FIN de Específicos de Microsoft

Vea también

[_com_error \(clase\)](#)

_com_error::GUID

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Llama a la función `IErrorInfo::GetGUID`.

Sintaxis

```
GUID GUID( ) const throw( );
```

Valor devuelto

Devuelve el resultado de `IErrorInfo::GetGUID` para el `IErrorInfo` objeto registrado en el `_com_error` objeto. Si no `IErrorInfo` se registra ningún objeto, devuelve `GUID_NULL`.

Observaciones

Se omite cualquier error que se produzca mientras se llama al `IErrorInfo::GetGUID` método.

FIN de Específicos de Microsoft

Vea también

[_com_error \(clase\)](#)

_com_error::HelpContext

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Llama a la función `IErrorInfo::GetHelpContext`.

Sintaxis

```
DWORD HelpContext( ) const throw( );
```

Valor devuelto

Devuelve el resultado de `IErrorInfo::GetHelpContext` para el `IErrorInfo` objeto registrado en el `_com_error` objeto. Si no `IErrorInfo` se registra ningún objeto, devuelve un cero.

Observaciones

Se omite cualquier error que se produzca mientras se llama al `IErrorInfo::GetHelpContext` método.

FIN de Específicos de Microsoft

Vea también

[_com_error \(clase\)](#)

_com_error::HelpFile

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Llama a la función `IErrorInfo::GetHelpFile`.

Sintaxis

```
_bstr_t HelpFile() const;
```

Valor devuelto

Devuelve el resultado de `IErrorInfo::GetHelpFile` para el `IErrorInfo` objeto registrado en el `_com_error` objeto. El BSTR resultante se encapsula en un objeto `_bstr_t`. Si no `IErrorInfo` se registra ningún, devuelve un vacío `_bstr_t`.

Observaciones

Se omite cualquier error que se produzca mientras se llama al `IErrorInfo::GetHelpFile` método.

FIN de Específicos de Microsoft

Vea también

[_com_error \(clase\)](#)

_com_error::HRESULTToWCode

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Asigna HRESULT de 32 bits a 16 bits `wCode`.

Sintaxis

```
static WORD HRESULTToWCode(
    HRESULT hr
) throw( );
```

Parámetros

h

HRESULT de 32 bits que se va a asignar a 16 bits `wCode`.

Valor devuelto

de 16 bits `wCode` asignado desde el HRESULT de 32 bits.

Observaciones

Vea [_com_error:: WCode](#) para obtener más información.

FIN de Específicos de Microsoft

Vea también

[_com_error::WCode](#)

[_com_error::WCodeToHRESULT](#)

[_com_error \(clase\)](#)

_com_error::Source

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Llama a la función `IErrorInfo::GetSource`.

Sintaxis

```
_bstr_t Source() const;
```

Valor devuelto

Devuelve el resultado de `IErrorInfo::GetSource` para el `IErrorInfo` objeto registrado en el `_com_error` objeto. El `BSTR` resultante se encapsula en un objeto `_bstr_t`. Si no `IErrorInfo` se registra ningún, devuelve un vacío `_bstr_t`.

Observaciones

Se omite cualquier error que se produzca mientras se llama al `IErrorInfo::GetSource` método.

FIN de Específicos de Microsoft

Vea también

[_com_error \(clase\)](#)

_com_error::WCode

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Recupera el código de error de 16 bits asignado al HRESULT encapsulado.

Sintaxis

```
WORD WCode ( ) const throw( );
```

Valor devuelto

Si el valor HRESULT está en el intervalo de 0x80040200 a 0x8004FFFF, el `wCode` método devuelve el HRESULT menos 0x80040200; de lo contrario, devuelve cero.

Observaciones

El `wCode` método se utiliza para deshacer una asignación que tiene lugar en el código de compatibilidad con com. El contenedor de una `dispinterface` propiedad o un método llama a una rutina de soporte que empaqueta los argumentos y las llamadas `IDispatch::Invoke`. Después de la devolución, si se devuelve un valor HRESULT de error `DISP_E_EXCEPTION`, la información de error se recupera de la `EXCEPINFO` estructura que se pasa a `IDispatch::Invoke`. El código de error puede ser un valor de 16 bits almacenado en el `wCode` miembro de la `EXCEPINFO` estructura o un valor completo de 32 bits en el `scode` miembro de la `EXCEPINFO` estructura. Si se devuelve un valor de 16 bits `wCode`, primero se debe asignar a un error HRESULT de 32 bits.

FIN de Específicos de Microsoft

Vea también

[_com_error::HRESULTToWCode](#)
[_com_error::WCodeToHRESULT](#)
[_com_error \(clase\)](#)

_com_error::WCodeToHRESULT

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Asigna *wCode* de 16 bits al HRESULT de 32 bits.

Sintaxis

```
static HRESULT WCodeToHRESULT(
    WORD wCode
) throw( );
```

Parámetros

wCode

WCode de 16 bits que se va a asignar al HRESULT de 32 bits.

Valor devuelto

HRESULT de 32 bits asignado desde el *wCode* de 16 bits.

Observaciones

Vea la función miembro [WCode](#).

FIN de Específicos de Microsoft

Vea también

[_com_error::WCode](#)

[_com_error::HRESULTToWCode](#)

[_com_error \(clase\)](#)

_com_error (Operadores)

06/03/2021 • 2 minutes to read • [Edit Online](#)

Para obtener información sobre los operadores de `_com_error` , vea [_com_error \(clase\)](#).

Consulta también

[_com_error \(clase\)](#)

_com_error::operator =

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Asigna un objeto `_com_error` existente a otro.

Sintaxis

```
_com_error& operator = (
    const _com_error& that
) throw ( );
```

Parámetros

that

Un objeto `_com_error`.

FIN de Específicos de Microsoft

Vea también

[_com_error \(clase\)](#)

_com_ptr_t (Clase)

06/03/2021 • 3 minutes to read • [Edit Online](#)

Específicos de Microsoft

Un objeto `_com_ptr_t` encapsula un puntero de interfaz com y se denomina puntero "inteligente". Esta clase de plantilla administra la asignación y desasignación de recursos a través de las llamadas de función a las `IUnknown` funciones miembro: `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:

```
_COM_SMARTPTR_TYPEDEF(IMyInterface, __uuidof(IMyInterface));
```

declara la especialización de `_com_ptr_t IMyInterfacePtr`.

Un conjunto de [plantillas de función](#), no miembros de esta clase de plantilla, admite comparaciones con un puntero inteligente en el lado derecho del operador de comparación.

Construcción

NOMBRE	DESCRIPCIÓN
<code>_com_ptr_t</code>	Construye un objeto de <code>_com_ptr_t</code> .

Operaciones de bajo nivel

NOMBRE	DESCRIPCIÓN
<code>AddRef</code>	Llama a la <code>AddRef</code> función miembro de <code>IUnknown</code> en el puntero de interfaz encapsulado.
<code>Adjuntar</code>	Encapsula un puntero de interfaz sin formato del tipo de este puntero inteligente.
<code>CreateInstance</code>	Crea una nueva instancia de un objeto, dado un <code>CLSID</code> o <code>ProgID</code> .
<code>Separar</code>	Extrae y devuelve el puntero de interfaz encapsulado.
<code>GetActiveObject</code>	Se adjunta a una instancia existente de un objeto, dado un <code>CLSID</code> o <code>ProgID</code> .
<code>GetInterfacePtr</code>	Devuelve el puntero de interfaz encapsulado.
<code>QueryInterface</code>	Llama a la <code>QueryInterface</code> función miembro de <code>IUnknown</code> en el puntero de interfaz encapsulado.
<code>Versión</code>	Llama a la <code>Release</code> función miembro de <code>IUnknown</code> en el puntero de interfaz encapsulado.

Operadores

NOMBRE	DESCRIPCIÓN
operador =	Asigna un nuevo valor a un objeto de <code>_com_ptr_t</code> existente.
operadores =,! =, <, > , <=, >=	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

Encabezado:<comip.h>

Lib: omsuppw.lib o comsuppwd.lib (vea [/Zc: Wchar_t \(Wchar_t es tipo nativo\)](#) para obtener más información)

Consulta también

[Clases de compatibilidad con COM del compilador](#)

`_com_ptr_t` (Funciones miembro)

06/03/2021 • 2 minutes to read • [Edit Online](#)

Para obtener información sobre las funciones miembro de `_com_ptr_t`, vea [_com_ptr_t \(clase\)](#).

Consulta también

[_com_ptr_t \(clase\)](#)

_com_ptr_t::_com_ptr_t

06/03/2021 • 3 minutes to read • [Edit Online](#)

Específicos de Microsoft

Construye un objeto de _com_ptr_t .

Sintaxis

```
// 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 dwClssContext = CLSCTX_ALL  
);  
  
// Calls CoCreateClass with provided CLSID retrieved from string.  
explicit _com_ptr_t(
```

```

LPCWSTR str,
IUnknown* pOuter = NULL,
DWORD dwClsContext = CLSCTX_ALL
);

// Constructs a smart pointer given a multibyte character string that
// holds either a CLSID (starting with "{") or a ProgID. This function
// calls CoCreateInstance, by the member function CreateInstance, to
// create a new COM object and then queries for this smart pointer's
// interface type. If QueryInterface fails with an E_NOINTERFACE error,
// a NULL smart pointer is constructed.
explicit _com_ptr_t(
    LPCSTR str,
    IUnknown* pOuter = NULL,
    DWORD dwClsContext = CLSCTX_ALL
);

// Saves the interface.
template<>
_com_ptr_t(
    Interface* pInterface
) throw();

// Make sure correct ctor is called
template<>
_com_ptr_t(
    LPSTR str
);

// Make sure correct ctor is called
template<>
_com_ptr_t(
    LPWSTR str
);

// Constructs a smart pointer from a different smart pointer type or
// from a different raw interface pointer. QueryInterface is called to
// find an interface pointer of this smart pointer's type. If
// QueryInterface fails with an E_NOINTERFACE error, a NULL smart
// pointer is constructed.
template<typename _OtherIID>
_com_ptr_t(
    const _com_ptr_t<_OtherIID>& p
);

// Constructs a smart-pointer from any IUnknown-based interface pointer.
template<typename _InterfaceType>
_com_ptr_t(
    _InterfaceType* p
);

// Disable conversion using _com_ptr_t* specialization of
// template<typename _InterfaceType> _com_ptr_t(_InterfaceType* p)
template<>
explicit _com_ptr_t(
    _com_ptr_t* p
);

```

Parámetros

pInterface

Puntero a interfaz sin formato.

fAddRef

Si `true` `AddRef` es, se llama a para incrementar el recuento de referencias del puntero de interfaz encapsulado.

cp

Objeto `_com_ptr_t`.

m

Puntero de interfaz sin formato, cuyo tipo es diferente del tipo de puntero inteligente de este objeto `_com_ptr_t`.

varSrc

Un objeto `_variant_t`.

`CLSID`

`CLSID` De una coclase.

dwClsContext

Contexto para el código ejecutable.

lpcStr

Cadena multibyte que contiene un (a `CLSID` partir de "{") o un `ProgID`.

pOuter

El externo desconocido para la [agregación](#).

FIN de Específicos de Microsoft

Vea también

[_com_ptr_t \(clase\)](#)

_com_ptr_t::AddRef

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Llama a la `AddRef` función miembro de `IUnknown` en el puntero de interfaz encapsulado.

Sintaxis

```
void AddRef( );
```

Observaciones

Llama a `IUnknown::AddRef` en el puntero de interfaz encapsulado y genera un `E_POINTER` error si el puntero es NULL.

FIN de Específicos de Microsoft

Vea también

[_com_ptr_t \(clase\)](#)

_com_ptr_t::Attach

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Encapsula un puntero de interfaz sin formato del tipo de este puntero inteligente.

Sintaxis

```
void Attach( Interface* pInterface ) throw( );
void Attach( Interface* pInterface, bool fAddRef ) throw( );
```

Parámetros

pInterface

Puntero a interfaz sin formato.

fAddRef

Si es `true`, `AddRef` se llama a. Si es así `false`, el `_com_ptr_t` objeto toma la propiedad del puntero de interfaz sin formato sin llamar a `AddRef`.

Observaciones

- **Adjuntar (*pInterface*)** `AddRef` no se llama a. La propiedad de la interfaz se pasa a este objeto `_com_ptr_t`. `Release` se llama a para reducir el recuento de referencias del puntero encapsulado previamente.
- **Adjuntar (*pInterface*, *fAddRef*)** Si *fAddRef* es `true`, `AddRef` se llama a para incrementar el recuento de referencias para el puntero de interfaz encapsulado. Si *fAddRef* es `false`, este `_com_ptr_t` objeto toma la propiedad del puntero de interfaz sin formato sin llamar a `AddRef`. `Release` se llama a para reducir el recuento de referencias del puntero encapsulado previamente.

FIN de Específicos de Microsoft

Vea también

[_com_ptr_t \(clase\)](#)

_com_ptr_t::CreateInstance

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Crea una nueva instancia de un objeto, dado un `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

`CLSID` De un objeto.

clsidString

Cadena Unicode que contiene un (a `CLSID` partir de "{") o un `ProgID`.

clsidStringA

Una cadena multibyte, mediante la página de códigos ANSI, que contiene `CLSID` (empezando por "{") o `ProgID`.

dwClsContext

Contexto para el código ejecutable.

pOuter

El externo desconocido para la [agregación](#).

Observaciones

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`. `Release` se llama a para reducir el recuento de referencias del puntero encapsulado previamente. Esta rutina devuelve HRESULT para indicar si se ha realizado correctamente o no.

- **CreateInstance (*rclsid*, *dwClsContext*)** Crea una nueva instancia en ejecución de un objeto, dado un `CLSID`.
- **CreateInstance (*clsidString*, *dwClsContext*)** Crea una nueva instancia en ejecución de un objeto, dada una cadena Unicode que contiene `CLSID` (a partir de "{") o `ProgID`.

- **CreateInstance** (*clsidStringA*, *dwClContext*) Crea una nueva instancia en ejecución de un objeto a partir de una cadena de caracteres multibyte que contiene **CLSID** (a partir de "{") o **ProgID** . Llama a **MultiByteToWideChar**, que supone que la cadena está en la página de códigos ANSI en lugar de en una página de códigos OEM.

FIN de Específicos de Microsoft

Vea también

[_com_ptr_t \(clase\)](#)

_com_ptr_t::Detach

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Extrae y devuelve el puntero de interfaz encapsulado.

Sintaxis

```
Interface* Detach( ) throw( );
```

Observaciones

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. Depende de usted llamar a `Release` en el puntero de interfaz devuelto.

FIN de Específicos de Microsoft

Vea también

[_com_ptr_t \(clase\)](#)

_com_ptr_t::GetActiveObject

06/03/2021 • 2 minutes to read • [Edit Online](#)

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

`CLSID` De un objeto.

clsidString

Cadena Unicode que contiene un (a `CLSID` partir de "{") o un `ProgID`.

clsidStringA

Una cadena multibyte, mediante la página de códigos ANSI, que contiene `CLSID` (empezando por "{") o `ProgID`

Observaciones

Estas funciones miembro llaman a `GetActiveObject` para recuperar un puntero a un objeto en ejecución que se ha registrado con OLE y, a continuación, consulta para el tipo de interfaz de este puntero inteligente. El puntero resultante se encapsula dentro de este objeto `_com_ptr_t`. `Release` se llama a para reducir el recuento de referencias del puntero encapsulado previamente. Esta rutina devuelve `HRESULT` para indicar si se ha realizado correctamente 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 `CLSID` (a partir de "{") o `ProgID`.
- `GetActiveObject (clsidStringA)` se adjunta a una instancia existente de un objeto, dada una cadena de caracteres multibyte que contiene `CLSID` (a partir de "{") o `ProgID`. Llama a `MultiByteToWideChar`, que supone que la cadena está en la página de códigos ANSI en lugar de en una página de códigos OEM.

FIN de Específicos de Microsoft

Vea también

[_com_ptr_t \(clase\)](#)

_com_ptr_t::GetInterfacePtr

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Devuelve el puntero de interfaz encapsulado.

Sintaxis

```
Interface* GetInterfacePtr( ) const throw( );
Interface*& GetInterfacePtr() throw();
```

Observaciones

Devuelve el puntero de interfaz encapsulado, que puede ser NULL.

FIN de Específicos de Microsoft

Vea también

[_com_ptr_t \(clase\)](#)

_com_ptr_t::QueryInterface

06/03/2021 • 2 minutes to read • [Edit Online](#)

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

suscripto

`IID` de un puntero de interfaz.

m

Puntero de interfaz sin formato.

Observaciones

Llama a `IUnknown::QueryInterface` en el puntero de interfaz encapsulado con el especificado `IID` y devuelve el puntero de interfaz sin formato resultante en *p*. Esta rutina devuelve HRESULT para indicar si se ha realizado correctamente o no.

FIN de Específicos de Microsoft

Vea también

[_com_ptr_t \(clase\)](#)

_com_ptr_t::Release

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Llama a la función miembro **Release** de `IUnknown` en el puntero de interfaz encapsulado.

Sintaxis

```
void Release();
```

Observaciones

Llama a `IUnknown::Release` en el puntero de interfaz encapsulado y genera un `E_POINTER` error si este puntero de interfaz es NULL.

FIN de Específicos de Microsoft

Vea también

[_com_ptr_t \(clase\)](#)

`_com_ptr_t` (Operadores)

06/03/2021 • 2 minutes to read • [Edit Online](#)

Para obtener información sobre los `_com_ptr_t` operadores, vea [clase `_com_ptr_t`](#).

Consulta también

[_com_ptr_t \(clase\)](#)

_com_ptr_t::operator =

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 );
```

Observaciones

Asigna un puntero de interfaz a este objeto `_com_ptr_t`.

FIN de Específicos de Microsoft

Vea también

[_com_ptr_t \(clase\)](#)

_com_ptr_t (Operadores relacionales)

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 );
```

Observaciones

Compara el objeto de puntero inteligente a otro puntero inteligente, puntero de interfaz sin formato o NULL. A excepción de las pruebas de puntero NULL, estos operadores consultan primero ambos punteros para `IUnknown` y comparan los resultados.

FIN de Específicos de Microsoft

Vea también

[_com_ptr_t \(clase\)](#)

`_com_ptr_t` Extractores

02/11/2020 • 2 minutes to read • [Edit Online](#)

Específico de Microsoft

Extrae el puntero de interfaz COM encapsulado.

Sintaxis

```
operator Interface*( ) const throw( );
operator Interface&( ) const;
Interface& operator*( ) const;
Interface* operator->( ) const;
Interface** operator&( ) throw( );
operator bool( ) const throw( );
```

Observaciones

- `operator Interface*` Devuelve el puntero de interfaz encapsulado, que puede ser NULL.
- `operator Interface&` Devuelve una referencia al puntero de interfaz encapsulado y genera un error si el puntero es NULL.
- `operator*` Permite que un objeto de puntero inteligente actúe como si fuera la interfaz encapsulada real al desreferenciarse.
- `operator->` Permite que un objeto de puntero inteligente actúe como si fuera la interfaz encapsulada real al desreferenciarse.
- `operator&` Libera cualquier puntero de interfaz encapsulado, reemplazándolo por NULL y devuelve la dirección del puntero encapsulado. Este operador permite pasar el puntero inteligente por dirección a una función que tiene un parámetro de *salida* 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.

NOTE

Dado `operator bool` que no se declara como `explicit`, `_com_ptr_t` se pueden convertir implícitamente a `bool`, que se pueden convertir a cualquier tipo escalar. Esto puede tener consecuencias inesperadas en el código. Habilite la [Advertencia del compilador \(nivel 4\) C4800](#) para evitar el uso involuntario de esta conversión.

Consulte también

[_com_ptr_t \(clase\)](#)

Plantillas de funciones relacionales

06/03/2021 • 2 minutes to read • [Edit Online](#)

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.

m

Un puntero inteligente.

Observaciones

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

Vea también

[_com_ptr_t \(clase\)](#)

_variant_t (Clase)

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Un objeto _variant_t encapsula el `VARIANT` tipo de datos. La clase administra la asignación y desasignación de recursos y realiza las llamadas de función a `VariantInit` y `VariantClear` según corresponda.

Construcción

NOMBRE	DESCRIPCIÓN
<code>_variant_t</code>	Construye un objeto de <code>_variant_t</code> .

Operaciones

NOMBRE	DESCRIPCIÓN
<code>Adjuntar</code>	Adjunta un <code>VARIANT</code> objeto al objeto <code>_variant_t</code> .
<code>Borrar</code>	Borra el objeto encapsulado <code>VARIANT</code> .
<code>ChangeType</code>	Cambia el tipo del objeto <code>_variant_t</code> al indicado <code>VARTYPE</code> .
<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 de <code>_variant_t</code> existente.
<code>operador = =,! =</code>	Compara dos objetos <code>_variant_t</code> para buscar igualdad o desigualdad.
<code>Extractores</code>	Extrae datos del objeto encapsulado <code>VARIANT</code> .

FIN de Específicos de Microsoft

Requisitos

Encabezado: <comutil.h>

Lib: omsuppw.lib o comsuppwd.lib (vea [/Zc: Wchar_t \(Wchar_t es tipo nativo\)](#) para obtener más información)

Consulta también

[Clases de compatibilidad con COM del compilador](#)

`_variant_t` (Funciones miembro)

06/03/2021 • 2 minutes to read • [Edit Online](#)

Para obtener información sobre las funciones miembro de `_variant_t`, vea [_variant_t \(clase\)](#).

Consulta también

[_variant_t \(clase\)](#)

_variant_t::_variant_t

06/03/2021 • 8 minutes to read • [Edit Online](#)

Específicos de Microsoft

Construye un objeto `_variant_t`.

Sintaxis

```
_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 `VARIANT` objeto que se va a copiar en el nuevo `_variant_t` objeto.

var_t_Src

Un objeto `_variant_t` que se va a copiar en el nuevo objeto `_variant_t`.

fCopy

Si `false` es, el `VARIANT` objeto proporcionado se adjunta al nuevo `_variant_t` objeto sin crear una nueva copia de `VariantCopy`.

iSrc, sSrc

Un valor entero que se va a copiar en el nuevo objeto `_variant_t`.

vtSrc

Del `VARTYPE` nuevo `_variant_t` objeto.

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

`bool` Valor que se va a copiar en el nuevo `_variant_t` objeto.

pIUnknownSrc

Puntero de interfaz COM a un objeto VT_UNKNOWN que se va a encapsular en el nuevo `_variant_t` objeto.

pDispSrc

Puntero de interfaz COM a un objeto VT_DISPATCH que se va a encapsular en el nuevo `_variant_t` objeto.

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

`char` Valor que se va a copiar en el nuevo `_variant_t` objeto.

usSrc

`unsigned short` Valor que se va a copiar en el nuevo `_variant_t` objeto.

ulSrc

`unsigned long` Valor que se va a copiar en el nuevo `_variant_t` objeto.

iSrc

`int` Valor que se va a copiar en el nuevo `_variant_t` objeto.

uiSrc

`unsigned int` Valor que se va a copiar en el nuevo `_variant_t` objeto.

i8Src

`__int64` Valor que se va a copiar en el nuevo `_variant_t` objeto.

ui8Src

Un valor de `__int64 sin signo` que se va a copiar en el nuevo `_variant_t` objeto.

Observaciones

- `_variant_t ()` Construye un objeto vacío `_variant_t`, `VT_EMPTY`.
- `_variant_t (variant& varSrc***)*` Construye un `_variant_t` objeto a partir de una copia del `VARIANT`

objeto. El tipo variant se conserva.

- `_variant_t (Variant * pVarSrc***)`* construye un `_variant_t` objeto a partir de una copia del `VARIANT` objeto. El tipo variant se conserva.
- `_variant_t (_variant_t& var_t_Src***)`* Construye un `_variant_t` objeto a partir de otro `_variant_t` objeto. El tipo variant se conserva.
- `_variant_t (Variant& varSrc, bool fCopy)` construye un `_variant_t` objeto a partir de un `VARIANT` objeto existente. Si `fCopy` es `false`, el objeto `Variant` se adjunta al nuevo objeto sin crear una copia.
- `*_variant_t (Short***sSrc, VARTYPE vtSrc = VT_I2)` construye un `_variant_t` objeto de tipo `VT_I2` o `VT_BOOL` a partir de un `short` valor entero. Cualquier otro `VARTYPE` resultado es un error de `E_INVALIDARG`.
- `_variant_t (Long lSrc, VARTYPE vtSrc = VT_I4)` construye un `_variant_t` objeto de tipo `VT_I4`, `VT_BOOL` o `VT_ERROR` a partir de un `long` valor entero. Cualquier otro `VARTYPE` resultado es un error de `E_INVALIDARG`.
- `_variant_t (float fltSrc)` construye un `_variant_t` objeto de tipo `VT_R4` a partir de un `float` valor numérico.
- `_variant_t (Double dblSrc, VARTYPE vtSrc = VT_R8)` construye un `_variant_t` objeto de tipo `VT_R8` o `VT_DATE` a partir de un `double` valor numérico. Cualquier otro `VARTYPE` resultado es un error de `E_INVALIDARG`.
- `_variant_t (CY& cySrc)` construye un `_variant_t` objeto de tipo `VT_CY` a partir de un `CY` objeto.
- `_variant_t (_bstr_t& bstrSrc)` construye un `_variant_t` objeto de tipo `VT_BSTR` a partir de un `_bstr_t` objeto. Se asigna un nuevo `BSTR`.
- `_variant_t (wchar_t * wstrSrc)` construye un `_variant_t` objeto de tipo `VT_BSTR` a partir de una cadena Unicode. Se asigna un nuevo `BSTR`.
- `_variant_t (char * strSrc)` Construye un `_variant_t` objeto de tipo `VT_BSTR` a partir de una cadena. Se asigna un nuevo `BSTR`.
- `_variant_t (bool bSrc)` construye un `_variant_t` objeto de tipo `VT_BOOL` a partir de un `bool` valor.
- `_variant_t (IUnknown * pIUnknownSrc, bool fAddRef = true)` Construye un `_variant_t` objeto de tipo `VT_UNKNOWN` a partir de un puntero de interfaz com. Si `fAddRef` es `true`, `AddRef` se llama a en el puntero de interfaz proporcionado para que coincida con la llamada a `Release` que se producirá cuando `_variant_t` se destruya el objeto. Depende de usted llamar a `Release` en el puntero de interfaz proporcionado. Si `fAddRef` es `false`, este constructor toma la propiedad del puntero de interfaz proporcionado; no llama a `Release` en el puntero de interfaz proporcionado.
- `_variant_t (IDispatch * pDispSrc, bool fAddRef = true)` Construye un `_variant_t` objeto de tipo `VT_DISPATCH` a partir de un puntero de interfaz com. Si `fAddRef` es `true`, `AddRef` se llama a en el puntero de interfaz proporcionado para que coincida con la llamada a `Release` que se producirá cuando `_variant_t` se destruya el objeto. Depende de usted llamar a `Release` en el puntero de interfaz proporcionado. Si `fAddRef` es `false`, este constructor toma la propiedad del puntero de interfaz proporcionado; no llama a `Release` en el puntero de interfaz proporcionado.
- `_variant_t (decimal& decSrc)` construye un `_variant_t` objeto de tipo `VT_DECIMAL` a partir de un `DECIMAL` valor.
- `_variant_t (byte bSrc)` construye un `_variant_t` objeto de tipo `VT_UI1` a partir de un `BYTE` valor.

FIN de Específicos de Microsoft

Vea también

[_variant_t \(clase\)](#)

_variant_t::Attach

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Adjunta un `VARIANT` objeto al objeto `_variant_t`.

Sintaxis

```
void Attach(VARIANT& varSrc);
```

Parámetros

varSrc

`VARIANT` Objeto que se va a adjuntar a este objeto `_variant_t`.

Observaciones

Toma la propiedad de `VARIANT` mediante su encapsulado. Esta función miembro libera cualquier encapsulado existente `VARIANT` y, a continuación, copia el proporcionado `VARIANT`, y establece su `VARTYPE` en `VT_EMPTY` para asegurarse de que sus recursos solo pueden ser liberados por el `_variant_t` destructor.

FIN de Específicos de Microsoft

Vea también

[_variant_t \(clase\)](#)

_variant_t::Clear

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Borra el objeto encapsulado `VARIANT`.

Sintaxis

```
void Clear( );
```

Observaciones

Llama a `VariantClear` en el objeto encapsulado `VARIANT`.

FIN de Específicos de Microsoft

Vea también

[_variant_t \(clase\)](#)

_variant_t::ChangeType

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Cambia el tipo del `_variant_t` objeto al indicado `VARTYPE`.

Sintaxis

```
void ChangeType(
    VARTYPE vartype,
    const _variant_t* pSrc = NULL
);
```

Parámetros

VarType

`VARTYPE` Para este `_variant_t` objeto.

pSrc

Un puntero al objeto `_variant_t` que se va a convertir. Si este valor es NULL, la conversión se realiza en contexto.

Observaciones

Esta función miembro convierte un `_variant_t` objeto en el indicado `VARTYPE`. Si *pSrc* es null, la conversión se realiza en contexto; de lo contrario, este `_variant_t` objeto se copia de *pSrc* y, a continuación, se convierte.

FIN de Específicos de Microsoft

Vea también

[_variant_t \(clase\)](#)

_variant_t::Detach

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Desasocia el objeto encapsulado `VARIANT` de este `_variant_t` objeto.

Sintaxis

```
VARIANT Detach( );
```

Valor devuelto

Encapsulado `VARIANT`.

Observaciones

Extrae y devuelve el encapsulado y, `VARIANT` a continuación, borra este `_variant_t` objeto sin destruirlo. Esta función miembro quita `VARIANT` de la encapsulación de y establece el `VARTYPE` de este `_variant_t` objeto en `VT_EMPTY`. Depende de usted que libere el devuelto `VARIANT` mediante una llamada a la función [VariantClear](#).

FIN de Específicos de Microsoft

Vea también

[_variant_t \(clase\)](#)

_variant_t::SetString

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Asigna una cadena a este objeto `_variant_t`.

Sintaxis

```
void SetString(const char* pSrc);
```

Parámetros

pSrc

Puntero a la cadena de caracteres.

Observaciones

Convierte una cadena de caracteres ANSI en una cadena `BSTR` Unicode y la asigna a este objeto `_variant_t`.

FIN de Específicos de Microsoft

Vea también

[_variant_t \(clase\)](#)

`_variant_t` (Operadores)

06/03/2021 • 2 minutes to read • [Edit Online](#)

Para obtener información sobre los operadores de `_variant_t`, vea [_variant_t \(clase\)](#).

Consulta también

[_variant_t \(clase\)](#)

`_variant_t::operator=`

30/03/2021 • 5 minutes to read • [Edit Online](#)

Asigna un nuevo valor a una instancia de `_variant_t`.

La `_variant_t` clase y su `operator=` miembro son **específicos de Microsoft**.

Sintaxis

```
_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 `VT_*` tipo.

`pVarSrc`

Un puntero a un `VARIANT` desde el que se va a copiar el contenido y el `VT_*` tipo.

`var_t_Src`

Referencia a un `_variant_t` desde el que se va a copiar el contenido y el `VT_*` tipo.

`sSrc`

`short` Valor entero que se va a copiar. Tipo especificado `VT_BOOL` si `*this` es de tipo `VT_BOOL`. De lo contrario, se proporciona el tipo `VT_I2`.

`lSrc`

`long` Valor entero que se va a copiar. Tipo especificado `VT_BOOL` si `*this` es de tipo `VT_BOOL`. Tipo especificado `VT_ERROR` si `*this` es de tipo `VT_ERROR`. De lo contrario, se especifica el tipo `VT_I4`.

`fltSrc`

`float` Valor numérico que se va a copiar. Tipo especificado `VT_R4`.

`dblSrc`

`double` Valor numérico que se va a copiar. Tipo especificado `VT_DATE` si `this` es de tipo `VT_DATE`. De lo contrario, se especifica el tipo `VT_R8`.

`cySrc`

Objeto `CY` que se va a copiar. Tipo especificado `VT_CY`.

`bstrSrc`

Objeto `BSTR` que se va a copiar. Tipo especificado `VT_BSTR`.

`wstrSrc`

Cadena Unicode que se va a copiar, almacenada como un `BSTR` tipo y un tipo determinado `VT_BSTR`.

`strSrc`

Cadena multibyte que se va a copiar, almacenada como un `BSTR` tipo y un tipo determinado `VT_BSTR`.

`pDispSrc`

`IDispatch` Puntero que se va a copiar con una llamada a `AddRef`. Tipo especificado `VT_DISPATCH`.

`bSrc`

`bool` Valor que se va a copiar. Tipo especificado `VT_BOOL`.

`pSrc`

`IUnknown` Puntero que se va a copiar con una llamada a `AddRef`. Tipo especificado `VT_UNKNOWN`.

`decSrc`

Objeto `DECIMAL` que se va a copiar. Tipo especificado `VT_DECIMAL`.

`byteSrc`

`BYTE` Valor que se va a copiar. Tipo especificado `VT_UI1`.

`cSrc`

`char` Valor que se va a copiar. Tipo especificado `VT_I1`.

`usSrc`

`unsigned short` Valor que se va a copiar. Tipo especificado `VT_UI2`.

`ulSrc`

`unsigned long` Valor que se va a copiar. Tipo especificado `VT_UI4`.

`iSrc`

`int` Valor que se va a copiar. Tipo especificado `VT_INT`.

`uiSrc`

`unsigned int` Valor que se va a copiar. Tipo especificado `VT_UINT`.

`i8Src`

`_int64` Valor o `long long` que se va a copiar. Tipo especificado `VT_I8`.

`ui8Src`

`unsigned _int64` Valor o `unsigned long long` que se va a copiar. Tipo especificado `VT_UI8`.

Comentarios

El `operator=` operador de asignación borra cualquier valor existente, que elimina los tipos de objeto o llama a `Release` para los `IDispatch*`, `IUnknown*` tipos y. Después, copia un nuevo valor en el `_variant_t` objeto. Cambia el `_variant_t` tipo para que coincida con el valor asignado, excepto como se indica en los `short`, `long` argumentos, y `double`. Los tipos de valor se copian directamente. Un `VARIANT` puntero o un `_variant_t` argumento de referencia copia el contenido y el tipo del objeto asignado. Otros argumentos de tipo de puntero

o de referencia crean una copia del objeto asignado. El operador de asignación llama a `AddRef` para los `IDispatch*` `IUnknown*` argumentos y.

`operator=` invoca `_com_raise_error` si se produce un error.

`operator=` Devuelve una referencia al objeto actualizado `_variant_t`.

Consulte también

[Clase `_variant_t`](#)

_variant_t (Operadores relacionales)

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 `_variant_t` objeto.

pSrc

Puntero a la `VARIANT` que se va a comparar con el `_variant_t` objeto.

Valor devuelto

Devuelve `true` si la comparación contiene, de lo `false` contrario.

Observaciones

Compara un `_variant_t` objeto con un objeto `VARIANT`, probando la igualdad o la desigualdad.

FIN de Específicos de Microsoft

Vea también

[_variant_t \(clase\)](#)

_variant_t (Extractores)

06/03/2021 • 2 minutes to read • [Edit Online](#)

Específicos de Microsoft

Extrae datos del objeto encapsulado `VARIANT`.

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;
```

Observaciones

Extrae datos sin procesar de un encapsulado `VARIANT`. Si el `VARIANT` todavía no es el tipo adecuado, `VariantChangeType` se utiliza para intentar una conversión y se genera un error en caso de error:

- **operador Short ()** Extrae un `short` valor entero.
- **operador Long ()** Extrae un `long` valor entero.
- **Operator Float ()** Extrae un `float` valor numérico.
- **Operator Double ()** Extrae un `double` valor entero.
- **operador CY ()** Extrae un `CY` objeto.
- **Operator bool ()** Extrae un `bool` valor.
- **Operator decimal ()** Extrae un `DECIMAL` valor.
- **operador byte ()** Extrae un `BYTE` valor.
- **operador _bstr_t ()** Extrae una cadena, que se encapsula en un `_bstr_t` objeto.
- el **operador IDispatch * ()** extrae un puntero dispinterface de un encapsulado `VARIANT`. `AddRef` se llama a en el puntero resultante, por lo que depende de usted llamar `Release` a para liberarlo.
- **Operator IUnknown * ()** extrae un puntero de interfaz com de un encapsulado `VARIANT`. `AddRef` se

Llama a en el puntero resultante, por lo que depende de usted llamar `Release` a para liberarlo.

FIN de Específicos de Microsoft

Vea también

[_variant_t \(clase\)](#)

Extensiones de Microsoft

06/03/2021 • 2 minutes to read • [Edit Online](#)

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 implementaciones futuras)  
__oldcall (reservado para implementaciones futuras)  
__unaligned (reservado para implementaciones futuras)  
based-modifier
```

based-modifier :

```
__based ( based-type )
```

based-type :

```
name
```

Comportamiento no estándar

06/03/2021 • 3 minutes to read • [Edit Online](#)

En las secciones siguientes se enumeran algunos de los lugares en los que la implementación de Microsoft de C++ no cumple con el 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 los [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 cumple con el párrafo 7 de la sección 10.3 de la especificación ISO de C++. El ejemplo siguiente no se compila y se produce el error del compilador [C2688](#)

```
// 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

El compilador de Microsoft C++ no admite actualmente el enlace de nombres no dependientes cuando se analiza inicialmente una plantilla. Esto no cumple con la sección 14.6.3 de la especificación ISO de C++. Esto puede hacer que se vean las sobrecargas declaradas después de la plantilla (pero antes de que se creen instancias de la plantilla).

```
#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 cumple con la sección 15.4 de la especificación ISO de C++. Por ejemplo:

```
void f() throw(int); // parsed but not used
void g() throw();    // parsed and used
```

Para obtener más información sobre las especificaciones de excepción, vea [Especificaciones de excepciones](#).

char_traits::eof()

Los Estados estándar de C++ que `char_traits::EOF` no deben corresponder a un `char_type` valor válido. El compilador de Microsoft C++ exige esta restricción para el tipo `char`, pero no para el tipo `wchar_t`. Esto no cumple con el requisito indicado en la tabla 62 de la sección 12.1.1 de la especificación ISO de C++. En el ejemplo siguiente se muestra esto.

```
#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 los tipos sin miembros de datos compartirán una ubicación de almacenamiento con otros tipos mientras dure el objeto.

Límites del compilador

06/03/2021 • 2 minutes to read • [Edit Online](#)

El estándar de C++ recomienda límites para varias construcciones del lenguaje. A continuación se muestra 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 instrucciones 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 instrucciones anidada, pero por lo general entre 100 y 110.
- Parámetros en una definición de macro: estándar de C++: 256, compilador de Microsoft C++: 127.
- Argumentos en una invocación de macros: estándar de C++: 256, compilador de Microsoft C++ 127.
- Caracteres en un literal de cadena de caracteres o literal de cadena ancho (después de la concatenación)-estándar de C++: 65536, compilador de Microsoft C++: 65535 caracteres de un solo byte, incluido el terminador nulo, y 32767 caracteres de doble byte, incluido el terminador NULL.
- Niveles de definiciones de clase anidada, estructura o unión en un `struct-declaration-list` estándar de C++ único: 256, compilador de Microsoft C++: 16.
- Inicializadores de miembro en una definición de constructor: estándar de C++: 6144, compilador de Microsoft C++: al menos 6144.
- Calificaciones de ámbito de un identificador: estándar de C++: 256, compilador de Microsoft C++: 127.
- Especificaciones anidadas: `extern` estándar de C++: 1024, compilador de Microsoft C++: 9 (sin contar la `extern` especificación implícita en el ámbito global, o 10, si se cuenta la `extern` especificación implícita en el ámbito global..
- Argumentos de plantilla en una declaración de plantilla: estándar de C++: 1024, compilador de Microsoft C++: 2046.

Consulta también

[Comportamiento no estándar](#)

Referencia del preprocesador de C/C++

02/11/2020 • 2 minutes to read • [Edit Online](#)

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: preprocesador` proporciona un preprocesador C11 y C17 totalmente compatible. Este es el valor predeterminado cuando se usa la marca del compilador `/std:c11` o `/std:c17`.

En esta sección

[Preprocesador](#)

Proporciona información general sobre los preprocesadores de conformidad tradicionales y nuevos.

[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 tal y como se especifica en los estándares de C y C++ y en 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 biblioteca estándar de C++

10/03/2021 • 2 minutes to read • [Edit Online](#)

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 acerca de la vinculación con el archivo de tiempo de ejecución de Visual C++ adecuado `.lib`, vea archivos de la [biblioteca .lib](#) estándar de C (CRT) y C++ (STL).

En esta sección

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

Proporciona información general sobre la implementación de Microsoft de la biblioteca estándar de C++.

`iostream` Programación

Proporciona información general sobre la `iostream` programación de.

[Referencia de archivos de encabezado](#)

Proporciona vínculos a temas de referencia sobre los archivos de encabezado de la biblioteca estándar de C++, con ejemplos de código.