Contents

- Referencia del preprocesador de C/C++
- Preprocesador
  - Preprocesador
  - Información general del nuevo preprocesador
  - Fases de traducción
  - Directivas de preprocesador
    - Directivas de preprocesador
    - #define (Directiva) (C/C++)
    - #error (Directiva) (C/C++)
    - #if, #elif, #elsey #endif (Directivas) (C/C++)
    - #ifdef e #ifndef (Directivas) (C/C++)
    - #import (Directiva) (C++)
    - #import (atributos) (C++)
      - #import (atributos) (C++)
      - auto_rename
      - auto_search
      - embedded_idl
      - exclude (#import)
      - high_method_prefix
      - high_property_prefixes
      - implementation_only
      - include()
      - inject_statement
      - named_guids
      - no_auto_exclude
      - no_dual_interfaces
      - no_implementation
      - no_namespace
      - no_registry

no_search_namespace
no_smart_pointers
raw_dispinterfaces
raw_interfaces_only
raw_method_prefix
raw_native_types
raw_property_prefixes
rename (#import)
rename_namespace
rename_search_namespace
tlbid
#include (Directiva) (C/C++)
#line (Directiva) (C/C++)
NULL (Directiva)
#undef (Directiva) (C/C++)
#using (directiva) (C++/CLI)
Operadores de preprocesador
Operadores de preprocesador
Operador de conversión a cadenas (#)
Operador de generación de caracteres (#@)
Operador de pegado de token (##)
Macros de preprocesador (C/C++)
Macros de preprocesador (C/C++)
Macros de preprocesador y C++
Macros de preprocesador variádicas
Macros de preprocesador predefinidas
Resumen de la gramática del preprocesador (C/C++)
Directivas pragma y la palabra clave __pragma
Directivas pragma y la palabra clave __pragma
alloc_text (pragma)
auto_inline (Pragma)
bss_seg (Pragma)

check_stack (Pragma)
code_seg (Pragma)
comment (Pragma)
component (Pragma)
conform (Pragma)
const_seg (Pragma)
data_seg (Pragma)
deprecated (Pragma) (C/C++)
detect_mismatch (Pragma)
execution_character_set (Pragma)
fenv_access (Pragma)
float_control (Pragma)
fp_contract (Pragma)
function (Pragma)
hdrstop (Pragma)
include_alias (Pragma)
init_seg (Pragma)
inline_depth (Pragma)
inline_recursion (Pragma)
intrinsic (Pragma)
loop (Pragma)
make_public (Pragma)
managed, unmanaged (Pragmas)
message (Pragma)
omp (Pragma)
once (Pragma)
optimize (Pragma)
pack (Pragma)
pointers_to_members (Pragma)
pop_macro (Pragma)
push_macro (Pragma)
region, endregion (Pragmas)

runtime_checks (Pragma)
section (Pragma)
setlocale (Pragma)
strict_gs_check (Pragma)
system_header (pragma)
tvtordisp (Pragma)
warning (pragma)

Referencia del preprocesador de C/C++

13/08/2021 • 2 minutes to read

La *referencia del preprocesador de C/C++* explica el preprocesador tal como se implementa en Microsoft C/C++. El preprocesador realiza operaciones preliminares en archivos de C y C++ antes de pasarlos al compilador. Puede usar el preprocesador para realizar las siguientes acciones de manera condicional: compilar código, insertar archivos, especificar mensajes de error en tiempo de compilación y aplicar reglas específicas del equipo a secciones del código.

En Visual Studio 2019, la opción del compilador /Zc:preprocessor proporciona un preprocesador C11 y C17 totalmente compatible. Este es el valor predeterminado cuando se usa la marca del compilador `/std:c11` o `/std:c17` .

En esta sección

Preprocesador
Proporciona información general sobre los preprocesadores conformes 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 según lo especificado por los estándares de C y C++ y por Microsoft C++.

Pragmas
Describe las directivas pragma, que proporcionan un método para que cada compilador ofrezca características específicas del equipo o del sistema operativo a la vez que conserva la compatibilidad total con los lenguajes C y C++.

Secciones relacionadas

Referencia del lenguaje C++
Proporciona material de referencia para la implementación de Microsoft del lenguaje C++.

Referencia del lenguaje C
Proporciona material de referencia para la implementación de Microsoft del lenguaje C.

Referencia de compilación de C/C++
Proporciona vínculos a temas que describen las opciones del compilador y el vinculador.

Visual Studio proyectos: 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++.

Preprocesador

12/08/2021 • 2 minutes to read

El preprocesador es un procesador de texto que manipula el texto de un archivo de código fuente como parte de la primera fase de traducción. El preprocesador no analiza el texto de origen, pero lo divide en tokens para buscar llamadas macro. Aunque el compilador invoca normalmente el preprocesador en el primer paso, también se puede invocar el preprocesador por separado para procesar el texto sin compilación.

El material de referencia del preprocesador incluye las siguientes secciones:

- Directivas de preprocesador
- Operadores de preprocesador
- Macros predefinidas
- Pragmas

Específicos de Microsoft

Puede obtener una lista del código fuente después del preprocesamiento mediante la opción del compilador /E o /EP. Ambas opciones invocan el preprocesador y envían el texto resultante al dispositivo de salida estándar, que, en la mayoría de los casos, es la consola. La diferencia entre las dos opciones es que incluye directivas y elimina `/E` `#line` estas `/EP` directivas.

FIN de Específicos de Microsoft

Terminología especial

En la documentación del preprocesador, el término "argumento" hace referencia a la entidad que se pasa a una función. En algunos casos, se modifica mediante "real" o "formal", que describe la expresión de argumento especificada en la llamada de función y la declaración de argumento especificada en la definición de función, respectivamente.

El término "variable" hace referencia a un objeto de datos de tipo C. El término "objeto" hace referencia a objetos y variables de C++; es un término inclusivo.

Consulte también

Referencia del preprocesador de C/C++
Fases de traducción

MSVC información general sobre el nuevo preprocesador

13/08/2021 • 8 minutes to read

Visual Studio 2015 usa el preprocesador tradicional, que no se ajusta a C++ estándar o C99. A partir Visual Studio versión 16.5 de 2019, la nueva compatibilidad con el preprocesador para el estándar C++20 está completa. Estos cambios están disponibles mediante el modificador del compilador /Zc:preprocessor. Hay disponible una versión experimental del nuevo preprocesador a partir de la versión 15.8 de Visual Studio 2017 y versiones posteriores mediante el modificador del compilador /experimental:preprocessor. Puede encontrar más información sobre el uso del nuevo preprocesador Visual Studio 2017 y Visual Studio 2019. Para ver la documentación de su versión preferida de Visual Studio, use el control de selector **Versión**. Se encuentra en la parte superior de la tabla de contenido de esta página.

Estamos actualizando el preprocesador de Microsoft C++ para mejorar la conformidad con los estándares, corregir errores de larga duración y cambiar algunos comportamientos que oficialmente no están definidos. También hemos agregado nuevos diagnósticos para advertir sobre errores en las definiciones de macro.

A partir Visual Studio versión 16.5 de 2019, la compatibilidad con el preprocesador para el estándar C++20 está completa. Estos cambios están disponibles mediante el modificador del compilador /Zc:preprocessor. Hay disponible una versión experimental del nuevo preprocesador en versiones anteriores a partir de Visual Studio 2017, versión 15.8. Puede habilitarlo mediante el modificador del compilador /experimental:preprocessor. El comportamiento predeterminado del preprocesador sigue siendo el mismo que en versiones anteriores.

Nueva macro predefinida

Puede detectar qué preprocesador está en uso en tiempo de compilación. Compruebe el valor de la macro predefinida `_MSVC_TRADITIONAL` para saber si el preprocesador tradicional está en uso. Esta macro se establece incondicionalmente por las versiones del compilador que la admiten, independientemente del preprocesador que se invoque. Su valor es 1 para el preprocesador tradicional. Es 0 para el preprocesador conforme.

```
#if defined(_MSVC_TRADITIONAL) && _MSVC_TRADITIONAL
// Logic using the traditional preprocessor
#else
// Logic using cross-platform compatible preprocessor
#endif
```

Cambios de comportamiento en el nuevo preprocesador

El trabajo inicial en el nuevo preprocesador se ha centrado en hacer que todas las expansiones de macro se ajusten al estándar. Permite usar el compilador MSVC con bibliotecas que actualmente están bloqueadas por los comportamientos tradicionales. Hemos probado el preprocesador actualizado en proyectos del mundo real. Estos son algunos de los cambios importantes más comunes que hemos encontrado:

Comentarios de macro

El preprocesador tradicional se basa en búferes de caracteres en lugar de tokens de preprocesador. Permite un comportamiento inusual, como el siguiente truco de comentario de preprocesador, que no funciona con el preprocesador conforme:

```
#if DISAPPEAR
#define DISAPPEARING_TYPE /##/
#else
#define DISAPPEARING_TYPE int
#endif

// myVal disappears when DISAPPEARING_TYPE is turned into a comment
DISAPPEARING_TYPE myVal;
```

La corrección conforme a los estándares es declarar `int myVal` dentro de las directivas `#ifdef/#endif` adecuadas:

```
#define MYVAL 1

#ifdef MYVAL
int myVal;
#endif
```

L#val

El preprocesador tradicional combina incorrectamente un prefijo de cadena con el resultado del operador de cadenas (#):

```
 #define DEBUG_INFO(val) L"debug prefix:" L#val
//                                       ^
//                                       this prefix

const wchar_t *info = DEBUG_INFO(hello world);
```

En este caso, el prefijo no es necesario porque los literales de cadena adyacentes se combinan después de la expansión de macro de `L` todos modos. La corrección compatible con versiones anteriores es cambiar la definición:

```
#define DEBUG_INFO(val) L"debug prefix:" #val
//                                       ^
//                                       no prefix
```

El mismo problema también se encuentra en las macros de conveniencia que "stringizen" el argumento en un literal de cadena ancho:

```
 // The traditional preprocessor creates a single wide string literal token
#define STRING(str) L#str
```

Puede corregir el problema de varias maneras:

- Use la concatenación de cadenas `L""` de y para agregar `#str` prefijo. Los literales de cadena adyacentes se combinan después de la expansión de macros:

  ```
  #define STRING1(str) L""#str
  ```

- Agregue el prefijo después de `#str` que esté en cadena con expansión de macro adicional.

  ```
  #define WIDE(str) L##str
  #define STRING2(str) WIDE(#str)
  ```

- Use el operador de concatenación `##` para combinar los tokens. El orden de las operaciones de y no se especifica, aunque todos los compiladores parecen evaluar `##` el operador antes en este `#` `#` `##` caso.

```
#define STRING3(str) L## #str
```

Advertencia sobre no válido ##

Cuando el operador de pegar tokens (##) no da como resultado un único token de preprocesamiento válido, el comportamiento es indefinido. El preprocesador tradicional no puede combinar los tokens de forma silenciosa. El nuevo preprocesador coincide con el comportamiento de la mayoría de los demás compiladores y emite un diagnóstico.

```
// The ## is unnecessary and does not result in a single preprocessing token.
#define ADD_STD(x) std::##x
// Declare a std::string
ADD_STD(string) s;
```

Elisión de comas en macros variádicas

El preprocesador MSVC elimina siempre las comas antes de los `__VA_ARGS__` reemplazos vacíos. El nuevo preprocesador sigue más de cerca el comportamiento de otros compiladores multiplataforma populares. Para que se quite la coma, debe faltar el argumento variádico (no solo vacío) y debe marcarse con un `##` operador . Considere el ejemplo siguiente:

```
void func(int, int = 2, int = 3);
// This macro replacement list has a comma followed by __VA_ARGS__
#define FUNC(a, ...) func(a, __VA_ARGS__)
int main()
{
    // In the traditional preprocessor, the
    // following macro is replaced with:
    // func(10,20,30)
    FUNC(10, 20, 30);

    // A conforming preprocessor replaces the
    // following macro with: func(1, ), which
    // results in a syntax error.
    FUNC(1, );
}
```

En el ejemplo siguiente, en la llamada al `FUNC2(1)` argumento variádico falta en la macro que se invoca. En la llamada al `FUNC2(1, )` argumento variádico está vacío, pero no falta (observe la coma en la lista de argumentos).

```
#define FUNC2(a, ...) func(a , ## __VA_ARGS__)
int main()
{
   // Expands to func(1)
   FUNC2(1);

   // Expands to func(1, )
   FUNC2(1, );
}
```

En el próximo estándar de C++20, este problema se ha solucionado agregando `__VA_OPT__` . La nueva compatibilidad con el preprocesador para está disponible a Visual Studio `__VA_OPT__` versión 16.5 de 2019.

Extensión de macro variádica de C++20

El nuevo preprocesador admite elisión de argumentos de macro variádicos de C++20:

```
#define FUNC(a, ...) __VA_ARGS__ + a
int main()
  {
  int ret = FUNC(0);
  return ret;
  }
```

Este código no se ajusta al estándar de C++20. En MSVC, el nuevo preprocesador extiende este comportamiento de C++20 a los modos estándar de lenguaje inferior (`/std:c++14` , `/std:c++17`). Esta extensión coincide con el comportamiento de otros compiladores principales de C++ multiplataforma.

Los argumentos de macro están "desempaquetados"

En el preprocesador tradicional, si una macro reenvía uno de sus argumentos a otra macro dependiente, el argumento no se "desempaquete" cuando se inserta. Normalmente, esta optimización pasa desapercibida, pero puede provocar un comportamiento inusual:

```
// Create a string out of the first argument, and the rest of the arguments.
#define TWO_STRINGS( first, ... ) #first, #__VA_ARGS__
#define A( ... ) TWO_STRINGS(__VA_ARGS__)
const char* c[2] = { A(1, 2) };

// Conforming preprocessor results:
// const char c[2] = { "1", "2" };

// Traditional preprocessor results, all arguments are in the first string:
// const char c[2] = { "1, 2", };
```

Al expandir , el preprocesador tradicional reenvía todos los argumentos empaquetados en al primer argumento de TWO_STRINGS, lo que deja vacío el argumento `A()` `__VA_ARGS__` `TWO_STRINGS` variádico de . Esto hace que el resultado `#first` de sea "1, 2" en lugar de simplemente "1". Si está siguiendo con atención, es posible que se pregunte qué ha ocurrido con el resultado de en la expansión tradicional del preprocesador: si el parámetro variádico está vacío, debería dar lugar a un literal de cadena `#__VA_ARGS__` `""` vacío. Un problema independiente ha hecho que no se genere el token literal de cadena vacío.

Volver a examinar la lista de reemplazo de macros

Después de reemplazar una macro, los tokens resultantes se examinan de nuevo en busca de identificadores de macro adicionales que reemplazar. El algoritmo utilizado por el preprocesador tradicional para realizar el nuevo análisis no se ajusta, como se muestra en este ejemplo basado en código real:

```
#define CAT(a,b) a ## b
#define ECHO(...) __VA_ARGS__
// IMPL1 and IMPL2 are implementation details
#define IMPL1(prefix,value) do_thing_one( prefix, value)
#define IMPL2(prefix,value) do_thing_two( prefix, value)

// MACRO chooses the expansion behavior based on the value passed to macro_switch
#define DO_THING(macro_switch, b) CAT(IMPL, macro_switch) ECHO(( "Hello", b))
DO_THING(1, "World");

// Traditional preprocessor:
// do_thing_one( "Hello", "World");
// Conforming preprocessor:
// IMPL1 ( "Hello","World");
```

Aunque este ejemplo puede parecer un poco contrived, lo hemos visto en código real.

Para ver lo que sucede, podemos dividir la expansión a partir de `DO_THING` :

1. `DO_THING(1, "World")` se expande a `CAT(IMPL, 1) ECHO(("Hello", "World"))`
2. `CAT(IMPL, 1)` se expande `IMPL ## 1` a , que se expande a `IMPL1`
3. Ahora los tokens están en este estado: `IMPL1 ECHO(("Hello", "World"))`
4. El preprocesador busca el identificador de macro de tipo función `IMPL1` . Puesto que no va seguido de , no se considera una invocación de macro de tipo `(` función.
5. El preprocesador pasa a los siguientes tokens. Busca que se invoca la macro de tipo `ECHO` función: `ECHO(("Hello", "World"))` , que se expande a . `("Hello", "World")`
6. `IMPL1` nunca se vuelve a considerar para la expansión, por lo que el resultado completo de las expansiones es: `IMPL1("Hello", "World");`

Para modificar la macro para que se comporte de la misma manera tanto en el nuevo preprocesador como en el preprocesador tradicional, agregue otra capa de direccionamiento indirecto:

```
#define CAT(a,b) a##b
#define ECHO(...) __VA_ARGS__
// IMPL1 and IMPL2 are macros implementation details
#define IMPL1(prefix,value) do_thing_one( prefix, value)
#define IMPL2(prefix,value) do_thing_two( prefix, value)
#define CALL(macroName, args) macroName args
#define DO_THING_FIXED(a,b) CALL( CAT(IMPL, a), ECHO(( "Hello",b)))
DO_THING_FIXED(1, "World");

// macro expands to:
// do_thing_one( "Hello", "World");
```

Características incompletas anteriores a la 16.5

A partir Visual Studio versión 16.5 de 2019, el nuevo preprocesador está completo para C++20. En versiones anteriores de Visual Studio, el nuevo preprocesador está principalmente completo, aunque alguna lógica de directiva de preprocesador todavía vuelve al comportamiento tradicional. Esta es una lista parcial de características incompletas en Visual Studio versiones anteriores a la 16.5:

- Compatibilidad con `_Pragma`
- Características de C++20
- Error de bloqueo de aumento: los operadores lógicos de expresiones constantes de preprocesador no se implementan completamente en el nuevo preprocesador antes de la versión 16.5. En algunas `#if` directivas, el nuevo preprocesador puede volver al preprocesador tradicional. El efecto solo es perceptible cuando se expanden las macros incompatibles con el preprocesador tradicional. Esto puede ocurrir al compilar ranuras de preprocesador Boost.

Fases de traducción

13/08/2021 • 2 minutes to read

Los programas de C y C++ constan de uno o más archivos de código fuente, cada uno de los cuales contiene parte del texto de programa. Un archivo de código fuente, junto con sus archivos de inclusión, archivos que se incluyen mediante la directiva de preprocesador, pero sin incluir secciones de código quitadas por directivas de compilación condicional como , se denomina unidad de `#include` `#if` *traducción*.

Los archivos de origen se pueden traducir en momentos diferentes. De hecho, es habitual traducir solo archivos no actualizados. Las unidades de traducción traducidas se pueden procesar en archivos de objetos o bibliotecas de código de objetos independientes. Estas unidades de traducción independientes traducidas se vinculan para formar un programa ejecutable o una biblioteca de vínculos dinámicos (DLL). Para obtener más información sobre los archivos que se pueden usar como entrada para el vinculador, vea Vincular archivos de entrada.

Las unidades de traducción pueden comunicarse mediante:

- Llamadas a funciones que tienen vinculación externa.
- Llamadas a funciones miembro de clase que tienen vinculación externa.
- Modificación directa de objetos que tienen vinculación externa.
- Modificación directa de archivos.
- Comunicación entre procesos (solo para aplicaciones basadas en Microsoft Windows).

La lista siguiente describe las fases en las que el compilador traduce los archivos:

Asignación de caracteres
Los caracteres del archivo de código fuente se asignan a la representación de código fuente interna. En esta fase, las secuencias de trígrafos se convierten en una representación interna de caracteres únicos.

Plicación de línea
Todas las líneas que terminan en una barra diagonal inversa ( ) inmediatamente seguidas de un carácter de nueva línea se unen con la línea siguiente en el archivo de origen, formando líneas lógicas a partir de las \ líneas físicas. A menos que esté vacío, un archivo de código fuente debe terminar en un carácter de nueva línea que no esté precedido de una barra diagonal inversa.

Tokenización
El archivo de código fuente se divide en tokens de preprocesamiento y caracteres de espacio en blanco. Cada uno de los comentarios del archivo de código fuente se reemplaza por un carácter de espacio. Los caracteres de nueva línea se conservan.

Preprocesamiento
Se ejecutan las directivas de preprocesamiento y se expanden las macros al archivo de código fuente. La instrucción `#include` invoca la traslación que comienza con los tres pasos anteriores de traslación en cualquier texto incluido.

Asignación de juego de caracteres
Todos los miembros y secuencias de escape de juego de caracteres de origen se convierten en sus equivalentes en el juego de caracteres de ejecución. Para Microsoft C y C++, tanto el juego de caracteres de ejecución de origen como el de ejecución son ASCII.

Concatenación de cadenas

Se concatenan todos los literales adyacentes de cadena y cadena de caracteres anchos. Por ejemplo, `"String " "concatenation"` se convierte en `"String concatenation"`.

Traducción
Todos los tokens se analizan sintáctica y semánticamente; estos tokens se convierten en código de objeto.

Acoplamiento
Se resuelven todas las referencias externas para crear un programa ejecutable o una biblioteca de vínculos dinámicos.

El compilador produce advertencias o errores durante las fases de traslación en las que encuentra errores de sintaxis.

El vinculador resuelve todas las referencias externas y crea un programa ejecutable o DLL combinando una o más unidades de traslación procesadas por separado junto con bibliotecas estándar.

Vea también

Preprocesador

Directivas de preprocesador

11/08/2021 • 2 minutes to read

Las directivas de preprocesador, como y , se usan normalmente para que los programas de origen sean fáciles de cambiar y fáciles de compilar `#define` `#ifdef` en diferentes entornos de ejecución. Las directivas del archivo de código fuente le dicen al preprocesador que tome medidas específicas. Por ejemplo, el preprocesador puede reemplazar tokens en el texto, insertar el contenido de otros archivos en el archivo de código fuente o suprimir la compilación de parte del archivo quitando secciones de texto. Las líneas de preprocesador se reconocen y se ejecutan antes de la expansión de macro. Por lo tanto, si una macro se expande en algo que parece un comando de preprocesador, el preprocesador no lo reconoce.

Las instrucciones de preprocesador usan el mismo juego de caracteres que las instrucciones de archivo de origen, con la excepción de que no se admiten secuencias de escape. El juego de caracteres utilizado en las instrucciones de preprocesador es el mismo que el juego de caracteres de la ejecución. El preprocesador también reconoce valores de caracteres negativos.

El preprocesador reconoce las siguientes directivas:

`#define`
`#elif`
`#else`
`#endif`

`#error`
`#if`
`#ifdef`
`#ifndef`

`#import`
`#include`
`#line`

`#pragma`
`#undef`
`#using`

El signo de número ( ) debe ser el primer carácter de espacio no blanco `#` en la línea que contiene la directiva . Los caracteres de espacio en blanco pueden aparecer entre el signo de número y la primera letra de la directiva. Algunas directivas incluyen argumentos o valores. Cualquier texto que siga a una directiva (excepto un argumento o valor que forma parte de la directiva) debe ir precedido del delimitador de comentario de una sola línea ( ) o entre delimitadores de comentario `//` (`/* */`). Las líneas que contienen directivas de preprocesador se pueden continuar precediendo inmediatamente al marcador de final de línea con una barra diagonal inversa (`\`).

Las directivas de preprocesador pueden aparecer en cualquier lugar de un archivo de código fuente, pero solo se aplican al resto del archivo de origen, una vez que aparecen.

Consulte también

Operadores de preprocesador
Macros predefinidas

#define directiva (C/C++)

16/08/2021 • 5 minutes to read

El **#define** crea una *macro*, que es la asociación de un identificador o identificador con parámetros con una cadena de token. Una vez definida la macro, el compilador puede sustituir la cadena de token para cada aparición del identificador del archivo de código fuente.

Sintaxis

```
#define token-string del identificador
#define identificador (identificador opt , ... , identifier opt ) token-string opt
```

Comentarios

La **#define** directiva hace que el compilador sustituya *la cadena de token* por cada aparición del *identificador* en el archivo de origen. El *identificador* solo se reemplaza cuando forma un token. Es decir, *el identificador* no se reemplaza si aparece en un comentario, en una cadena o como parte de un identificador más largo. Para obtener más información, vea Tokens.

El *argumento de cadena* de token consta de una serie de tokens, como palabras clave, constantes o instrucciones completas. Uno o varios caracteres de espacio en blanco deben separar *la cadena de token* del *identificador*. Este espacio en blanco no se considera parte de texto sustituido, ni tampoco cualquier espacio en blanco que vaya después del último token del texto.

Un `#define` sin una cadena de *token* quita las repeticiones del *identificador* del archivo de origen. El *identificador* permanece definido y se puede probar mediante las `#if defined` directivas y `#ifdef` .

El segundo formato de sintaxis define una macro de tipo función con parámetros. Este formato acepta una lista opcional de parámetros que deben aparecer entre paréntesis. Una vez definida la macro, cada aparición posterior de *identifier (identifier*$_{opt}$, ..., *identifier*$_{opt}$) se reemplaza por una versión del argumento de cadena de *token* que tiene argumentos reales sustituidos por parámetros formales.

Los nombres de parámetros formales aparecen *en la cadena de token* para marcar las ubicaciones donde se sustituyen los valores reales. Cada nombre de parámetro puede aparecer varias veces en *la cadena de token* y los nombres pueden aparecer en cualquier orden. El número de argumentos de la llamada debe coincidir con el número de parámetros en la definición de macro. El uso racional de paréntesis garantiza que los argumentos complejos se interpreten correctamente.

Los parámetros formales de la lista están separados por comas. Cada nombre de la lista debe ser único, y la lista se debe incluir entre paréntesis. Ningún espacio puede separar *el identificador* y el paréntesis de apertura. Use la concatenación de líneas , coloque una barra diagonal inversa ( ) inmediatamente antes del carácter de nueva línea , para directivas `\` largas en varias líneas de origen. El ámbito de un nombre de parámetro formal se extiende a la nueva línea que finaliza *la cadena de token*.

Cuando una macro se ha definido con el segundo formato de sintaxis, las instancias textuales subsiguientes seguidas de una lista de argumentos indican una llamada a macro. Los argumentos reales que siguen a una instancia de *identificador* en el archivo de origen coinciden con los parámetros formales correspondientes en la definición de macro. Cada parámetro formal de *la cadena de token* que no va precedida de un operador de cadena ( ), charizing ( ) o de pegar tokens ( ), o no seguido de un operador , se reemplaza por el argumento real `#` `#@` `##` `##` correspondiente. Cualquier macro en el argumento real se expande antes de que la directiva reemplace el parámetro formal. (Los operadores se describen en Operadores de preprocesador).

Los ejemplos siguientes de macros con argumentos ilustran la segunda forma de la **sintaxis #define** datos:

```
// Macro to define cursor lines
#define CURSOR(top, bottom) (((top) << 8) | (bottom))

// Macro to get a random integer with a specified range
#define getrandom(min, max) \
    ((rand()%(int)(((max) + 1)-(min)))+ (min))
```

Los argumentos con efectos secundarios a veces hacen que las macros den resultados inesperados. Un parámetro formal determinado puede aparecer más de una vez en *la cadena de token*. Si ese parámetro formal se sustituye por una expresión con efectos secundarios, la expresión, con sus efectos secundarios, se puede evaluar más de una vez. (Vea los ejemplos en Token-Pasting Operator (##) [Operador de pegar tokens (##)].

La directiva `#undef` hace que se olvide la definición de preprocesador de un identificador. Vea The #undef Directive para obtener más información.

Si el nombre de la macro que se define se produce en *la cadena de token* (incluso como resultado de otra expansión de macro), no se expande.

Un segundo **#define** para una macro con el mismo nombre genera una advertencia a menos que la segunda secuencia de token sea idéntica a la primera.

Específicos de Microsoft

Microsoft C/C++ permite volver a definir una macro si la nueva definición es sintácticamente idéntica a la definición original. Es decir, las dos definiciones pueden tener distintos nombres de parámetro. Este comportamiento difiere de ANSI C, que requiere que las dos definiciones sean léxicamente idénticas.

Por ejemplo, las dos macros siguientes son idénticas salvo en los nombres de parámetro. ANSI C no permite esta redefinición, pero Microsoft C/C++ lo compila sin errores.

```
#define multiply( f1, f2 ) ( f1 * f2 )
#define multiply( a1, a2 ) ( a1 * a2 )
```

Por otra parte, las dos macros siguientes no son idénticas y se generará una advertencia en Microsoft C/C++.

```
#define multiply( f1, f2 ) ( f1 * f2 )
#define multiply( a1, a2 ) ( b1 * b2 )
```

FIN de Específicos de Microsoft

En este ejemplo se muestra la **#define** directiva:

```
#define WIDTH       80
#define LENGTH      ( WIDTH + 10 )
```

La primera instrucción define el identificador `WIDTH` como la constante de tipo entero 80 y define `LENGTH` en términos de `WIDTH` y la constante de tipo entero 10. Cada aparición de `LENGTH` se reemplaza con (`WIDTH + 10`). A su vez, cada aparición de `WIDTH + 10` se reemplaza con la expresión (`80 + 10`). Los paréntesis alrededor de `WIDTH + 10` son importantes, porque controlan la interpretación en instrucciones como las siguientes:

```
var = LENGTH * 20;
```

Después de la fase de preprocesamiento, la instrucción se convierte en:

```
var = ( 80 + 10 ) * 20;
```

que se evalúa como 1800. Sin paréntesis, el resultado es:

```
var = 80 + 10 * 20;
```

que se evalúa como 280.

Específicos de Microsoft

Definir macros y constantes con la opción del compilador /D tiene el mismo efecto que usar una directiva **de** preprocesamiento #define al principio del archivo. Se pueden definir hasta 30 macros mediante la opción /D.

FIN de Específicos de Microsoft

Vea también

Directivas de preprocesador

#error directiva (C/C++)

14/08/2021 • 2 minutes to read

La **#error** emite un mensaje de error especificado por el usuario en tiempo de compilación y, a continuación, finaliza la compilación.

Sintaxis

```
#error token-string
```

Comentarios

El mensaje de error que emite esta directiva incluye el *parámetro token-string.* El *parámetro token-string* no está sujeto a la expansión de macros. Esta directiva es más útil durante el preprocesamiento, para notificar al desarrollador la incoherencia de un programa o la infracción de una restricción. En el ejemplo siguiente se muestra el procesamiento de errores durante el preprocesamiento:

```
#if !defined(__cplusplus)
#error C++ compiler required.
#endif
```

Vea también

Directivas de preprocesador

#if directivas #elif, #else y #endif (C/C++)

13/08/2021 • 5 minutes to read

La **#if,** con las directivas **#elif ,**#else y **#endif,** controla la compilación de partes de un archivo de código fuente. Si la expresión que escribe (después del #if **)** tiene un valor distinto de cero, el grupo de líneas inmediatamente después de la **directiva #if** se mantiene en la unidad de traducción.

Gramática

condicional :
if-part elif-parts$_{opt}$ *else-part*$_{opt}$ *endif-line*

if-part :
texto if-line

if-line :
#if *constant-expression*
#ifdef *identificador*
#ifndef *identificador*

elif-parts :
elif-line text
elif-parts elif-line text

elif-line :
#elif *constant-expression*

else-part :
texto de otra línea

else-line :
#else

endif-line :
#endif

Comentarios

Cada **#if** directiva de un archivo de código fuente debe coincidir con una **directiva** #endif cierre. Puede aparecer **cualquier #elif** directivas entre las directivas **#if** y **#endif,** pero como máximo se permite una #else directiva. La **#else** directiva, si está presente, debe ser la última directiva antes de **#endif**.

Las **#if**, **#elif**, **#else** y #endif pueden anidar en las partes de texto de **otras #if** directivas. Cada directiva **#else**, **#elif**, o **#endif** directiva pertenece a la directiva #if **anterior** más cercana.

Todas las directivas de compilación condicional, como **#if** y **#ifdef**, deben coincidir con una directiva #endif cierre antes del final del archivo. De lo contrario, se genera un mensaje de error. Cuando las directivas de compilación condicional están contenidas en archivos de inclusión, deben cumplir las mismas condiciones: no debe haber directivas de compilación condicional sin coincidencia al final del archivo de inclusión.

El reemplazo de macro se realiza dentro de la parte de la línea que sigue a un comando **#elif,** por lo que se puede usar una llamada macro en *la expresión constante*.

El preprocesador selecciona una de las repeticiones de texto dadas *para* su posterior procesamiento. Un bloque

especificado en *text* puede ser cualquier secuencia de texto. Puede ocupar más de una línea. Normalmente, *el* texto es texto del programa que tiene significado para el compilador o el preprocesador.

El preprocesador procesa el texto *seleccionado* y lo pasa al compilador. Si *el* texto contiene directivas de preprocesador, el preprocesador las lleva a cabo. Solo se compilan los bloques de texto seleccionados por el preprocesador.

El preprocesador selecciona un único elemento de texto evaluando la expresión constante que sigue a cada directiva **#if** o **#elif** hasta que encuentra una expresión constante verdadera (distinta de cero). Selecciona todo el texto (incluidas otras directivas de preprocesador que comienzan por) hasta su #elif **#** asociado, **#else** o **#endif**.

Si todas las apariciones de *constant-expression* son falsas o si **no** aparecen directivas #elif, el preprocesador selecciona el bloque de texto después de la **#else** anterior. Cuando no hay ninguna **cláusula #else** y todas las instancias de *constant-expression* del bloque **#if** son false, no se selecciona ningún bloque de texto.

Constant-expression es una expresión constante de entero con estas restricciones adicionales:

- Las expresiones deben tener un tipo entero y solo pueden incluir constantes de enteros, constantes de caracteres y **el operador** definido.
- La expresión no puede usar ni `sizeof` un operador de conversión de tipos.
- Es posible que el entorno de destino no pueda representar todos los intervalos de enteros.
- La traducción representa el `int` tipo de la misma manera que el tipo y del mismo modo que `long` `unsigned int` `unsigned long` .
- El traductor puede traducir constantes de caracteres a un conjunto de valores de código diferentes del conjunto para el entorno de destino. Para determinar las propiedades del entorno de destino, use una aplicación creada para ese entorno para comprobar los valores de *limits. Macros H.*
- La expresión no debe consultar el entorno y debe permanecer aislada de los detalles de implementación en el equipo de destino.

Operadores de preprocesador

definido

El operador de preprocesador **definido se** puede usar en expresiones constantes especiales, como se muestra en la sintaxis siguiente:

> **defined(** *identifier* **)**
> **identificador** *definido*

Esta expresión constante se considera true (distinto de cero) si *el identificador* está definido actualmente. De lo contrario, la condición es false (0). Un identificador definido como texto vacío se considera definido. El **operador definido** se puede usar en una directiva **#if** y una **#elif** directiva, pero no en ningún otro caso.

En el ejemplo siguiente, las **directivas #if** y **#endif** controlan la compilación de una de las tres llamadas de función:

```
#if defined(CREDIT)
    credit();
#elif defined(DEBIT)
    debit();
#else
    printerror();
#endif
```

La llamada de función a `credit` se compila si el identificador `CREDIT` está definido. Si el identificador `DEBIT` está definido, se compila la llamada de función a `debit`. Si no está definido ninguno de los identificadores, se compila la llamada a `printerror`. Y `CREDIT` son `credit` identificadores distintos en C y C++ porque sus casos son diferentes.

Las instrucciones de compilación condicional del ejemplo siguiente suponen una constante simbólica previamente definida denominada `DLEVEL`.

```
#if DLEVEL > 5
    #define SIGNAL  1
    #if STACKUSE == 1
        #define STACK   200
    #else
        #define STACK   100
    #endif
#else
    #define SIGNAL  0
    #if STACKUSE == 1
        #define STACK   100
    #else
        #define STACK   50
    #endif
#endif
#if DLEVEL == 0
    #define STACK 0
#elif DLEVEL == 1
    #define STACK 100
#elif DLEVEL > 5
    display( debugptr );
#else
    #define STACK 200
#endif
```

El primer **bloque #if** muestra dos conjuntos de directivas **anidadas #if**, **#else** y **#endif** directivas. El primer conjunto de directivas se procesa solo si `DLEVEL > 5` es true. De lo contrario, las instrucciones **#else** se procesan.

Las **#elif** directivas **#else** en el segundo ejemplo se usan para tomar una de las cuatro opciones, en función del valor de `DLEVEL`. La constante `STACK` se establece en 0, 100 o 200, según de la definición de `DLEVEL`. Si `DLEVEL` es mayor que 5, la instrucción

```
#elif DLEVEL > 5
display(debugptr);
```

se compila y `STACK` no se define.

Un uso común para la compilación condicional es evitar inclusiones múltiples del mismo archivo de encabezado. En C++, donde las clases se definen a menudo en archivos de encabezado, se pueden usar construcciones como esta para evitar varias definiciones:

```
/*  EXAMPLE.H - Example header file  */
#if !defined( EXAMPLE_H )
#define EXAMPLE_H

class Example
{
    //...
};

#endif // !defined( EXAMPLE_H )
```

El código anterior realiza comprobaciones para ver si se define la constante simbólica `EXAMPLE_H`. Si es así, el archivo ya se ha incluido y no es necesario volver a procesarlo. Si no, se define la constante `EXAMPLE_H` para marcar EXAMPLE.H como ya procesado.

__has_include

Visual Studio 2017, versión 15.3 y posteriores: determina si un encabezado de biblioteca está disponible para su inclusión:

```
#ifdef __has_include
#  if __has_include(<filesystem>)
#    include <filesystem>
#    define have_filesystem 1
#  elif __has_include(<experimental/filesystem>)
#    include <experimental/filesystem>
#    define have_filesystem 1
#    define experimental_filesystem
#  else
#    define have_filesystem 0
#  endif
#endif
```

Vea también

Directivas de preprocesador

`#ifdef` Directivas `#ifndef` y (C/C++)

13/08/2021 • 2 minutes to read

Las directivas de preprocesador y tienen el mismo efecto que la directiva cuando `#ifdef` se usa con el operador `#ifndef` `#if` `defined` .

Sintaxis

```
#ifdef identifier
#ifndef identifier
```

Estas directivas son equivalentes a:

```
#if defined identifier
#if !defined identifier
```

Comentarios

Puede usar las directivas `#ifdef` `#ifndef` y en cualquier `#if` lugar. La `#ifdef` *`identifier`* instrucción es equivalente a cuando se `#if 1` ha *`identifier`* definido . Es equivalente a cuando no se ha definido o no se ha `#if 0` definido por la directiva *`identifier`* `#undef` . Estas directivas solo comprueban la presencia o ausencia de identificadores definidos con `#define` , no comprueban los identificadores declarados en el código fuente de C o C++.

Estas directivas se proporcionan únicamente por compatibilidad con las versiones anteriores del lenguaje. Se `defined(` *`identifier`* `)` prefiere la expresión constante utilizada con `#if` la directiva .

La `#ifndef` directiva comprueba lo contrario de la condición comprobada por `#ifdef` . Si no se ha definido el identificador o si su definición se ha quitado con `#undef` , la condición es true (distinto de cero). De lo contrario, la condición es false (0).

Específicos de Microsoft

El *identificador* se puede pasar desde la línea de comandos mediante la `/D` opción . Se pueden especificar hasta 30 macros con `/D` .

La directiva es útil para comprobar si existe una definición, ya que se puede pasar una definición `#ifdef` desde la línea de comandos. Por ejemplo:

```
// ifdef_ifndef.CPP
// compile with: /Dtest /c
#ifndef test
#define final
#endif
```

FIN de Específicos de Microsoft

Vea también

Directivas de preprocesador

#import directiva (C++)

14/08/2021 • 7 minutes to read

Específicos de C++

Se utiliza para incorporar información de una biblioteca de tipos. El contenido de la biblioteca de tipos se convierte en clases de C++, que describen fundamentalmente las interfaces COM.

Sintaxis

```
#import "filename" [ attributes]
#import <filename> [ atributos]
```

Parámetros

Nombre
Especifica la biblioteca de tipos que se va a importar. El *nombre de* archivo puede ser uno de los siguientes tipos:

- El nombre de un archivo que contiene una biblioteca de tipos, como un archivo .olb, .tlb o .dll. La palabra clave , `file:` , puede preceder a cada nombre de archivo.
- El identificador de programa de un control en la biblioteca de tipos. La palabra clave , `progid:` , puede preceder a cada progid. Por ejemplo:

  ```
  #import "progid:my.prog.id.1.5"
  ```

  Para obtener más información sobre los progids, vea Especificar el identificador de localización y el número de versión.

  Cuando se usa un compilador cruzado de 32 bits en un sistema operativo de 64 bits, el compilador solo puede leer el subárbol del Registro de 32 bits. Es posible que desee utilizar el compilador de 64 bits nativo para compilar y registrar una biblioteca de tipos de 64 bits.
- El identificador de biblioteca de la biblioteca de tipos. La palabra clave , `libid:` , puede preceder a cada identificador de biblioteca. Por ejemplo:

  ```
  #import "libid:12341234-1234-1234-1234-123412341234" version("4.0") lcid("9")
  ```

  Si no especifica o , las `version` `lcid` reglas aplicadas a `progid:` también se aplican a `libid:` .
- Un archivo ejecutable (.exe).
- Un archivo de biblioteca (.dll) que contiene un recurso de biblioteca de tipos (por ejemplo, un archivo .ocx).
- Un documento compuesto que contiene una biblioteca de tipos.
- Cualquier otro formato de archivo que pueda entender la API **LoadTypeLib**.

Atributos
Uno o varios #import atributos. Separa los atributos con un espacio en blanco o con una coma. Por ejemplo:

```
#import "..\drawctl\drawctl.tlb" no_namespace, raw_interfaces_only
```

O bien

```
#import "..\drawctl\drawctl.tlb" no_namespace raw_interfaces_only
```

Comentarios

Orden de búsqueda del nombre de archivo

Filename va precedido opcionalmente de una especificación de directorio. El nombre de archivo debe designar un archivo existente. La diferencia entre las dos formas de sintaxis es el orden en que el preprocesador busca los archivos de biblioteca de tipos cuando no se especifica completamente la ruta de acceso.

FORMA DE SINTAXIS	ACCIÓN
Formato con comillas	Indica al preprocesador que busque primero los archivos de biblioteca de tipos en el directorio del archivo que contiene la instrucción #import y, **a** continuación, en los directorios de los archivos que incluyan ( ) ese `#include` archivo. A continuación, el preprocesador busca en las rutas de acceso que se muestran a continuación.
Formato con corchetes angulares	Indica al preprocesador que busque archivos de biblioteca de tipos en las siguientes rutas de acceso: 1. Lista de `PATH` rutas de acceso de variables de entorno 2. Lista de `LIB` rutas de acceso de variables de entorno 3. Ruta de acceso especificada por la opción del compilador /I, salvo que el compilador está buscando una biblioteca de tipos a la que se hace referencia desde otra biblioteca de tipos con el no_registry atributo .

Especificar el identificador de localización y el número de versión

Cuando especifica un identificador de programa, también puede especificar el identificador y el número de versión de localización del identificador de programa. Por ejemplo:

```
#import "progid:my.prog.id" lcid("0") version("4.0)
```

Si no especifica un identificador de localización, se elige un progid según las reglas siguientes:

- Si solo hay un identificador de localización, se usa uno.
- Si hay más de un identificador de localización, se usa el primero con el número de versión 0, 9 o 409.
- Si hay más de un identificador de localización y ninguno de ellos es 0, 9 o 409, se usa el último.
- Si no especifica un número de versión, se usa la versión más reciente.

Archivos de encabezado creados por la importación

#import crea dos archivos de encabezado que reconstruyen el contenido de la biblioteca de tipos en el código fuente de C++. El archivo de encabezado principal es similar al generado por el compilador Lenguaje de definición de interfaz de Microsoft (MIDL), pero con datos y código generados por el compilador adicionales. El archivo de encabezado principal tiene el mismo nombre base que la biblioteca de tipos, más un . Extensión TLH. El archivo de encabezado secundario tiene el mismo nombre base que la biblioteca de tipos, con una extensión

.TLI. Contiene implementaciones para funciones miembro generadas por el compilador y se incluye (`#include`) en el archivo de encabezado principal.

Si se importa una propiedad dispinterface que usa parámetros, #import no genera una instrucción `byref` __declspec(property) para la función.

Ambos archivos de encabezado se colocan en el directorio de salida especificado por la opción /Fo (archivo de objeto de nombre). A continuación, el compilador los lee y compila como si el archivo de encabezado principal se denominase mediante una `#include` directiva .

Las siguientes optimizaciones del compilador vienen con **la #import** directiva :

- El archivo de encabezado, cuando se crea, tiene la misma marca de tiempo que la biblioteca de tipos.
- Cuando **#import** se procesa, el compilador comprueba primero si el encabezado existe y está actualizado. En caso afirmativo, no es necesario volver a crearla.

La **#import** directiva también participa en la recompilación mínima y se puede colocar en un archivo de encabezado precompilado. Para obtener más información, vea Crear archivos de encabezado precompilados.

Archivo de encabezado de biblioteca de tipos principal

El archivo de encabezado principal de la biblioteca de tipos se compone de siete secciones:

- Encabezado reutilizable: consta de los comentarios, la instrucción `#include` para COMDEF.H (que define algunas macros estándar utilizadas en el encabezado) y otro tipo de información de instalación.
- Definiciones de tipo y referencias adelantadas: consta de declaraciones de estructura como `struct IMyInterface` y definiciones de tipo.
- Declaraciones de puntero inteligente: la clase de plantilla `_com_ptr_t` es un puntero inteligente. Encapsula punteros de interfaz y elimina la necesidad de llamar a `AddRef` las funciones , y `Release` `QueryInterface` . También oculta la llamada `CoCreateInstance` al crear un nuevo objeto COM. En esta sección se usa la instrucción macro para establecer definiciones de tipo de interfaces COM como especializaciones de plantilla de `_COM_SMARTPTR_TYPEDEF` la _com_ptr_t clase de plantilla. Por ejemplo, para la interfaz `IMyInterface` , . El archivo TLH contendrá:

  ```
  _COM_SMARTPTR_TYPEDEF(IMyInterface, __uuidof(IMyInterface));
  ```

  que el compilador expandirá a:

  ```
  typedef _com_ptr_t<_com_IIID<IMyInterface, __uuidof(IMyInterface)> > IMyInterfacePtr;
  ```

  El tipo `IMyInterfacePtr` se puede utilizar a continuación en lugar del puntero de interfaz sin formato `IMyInterface*` . Por lo tanto, no es necesario llamar a las distintas `IUnknown` funciones miembro
- Declaraciones typeinfo: consta principalmente de definiciones de clase y otros elementos que exponen los elementos typeinfo individuales devueltos por `ITypeLib:GetTypeInfo` . En esta sección, cada typeinfo de la biblioteca de tipos se refleja en el encabezado de forma dependiente de la información de `TYPEKIND` .
- Definición de GUID de estilo anterior opcional: contiene inicializaciones de las constantes de GUID con nombre. Estos nombres tienen el formato `CLSID_CoClass` y , similares a los `IID_Interface` generados por el compilador MIDL.
- Instrucción `#include` para el encabezado secundario de la biblioteca de tipos.

- Sección del pie de página reutilizable: actualmente incluye `#pragma pack(pop)`.

Todas las secciones, excepto la sección reutilizable de encabezado y la sección reutilizable de pie de página, se incluyen en un espacio de nombres con su nombre especificado por la instrucción en el `library` archivo IDL original. Puede usar los nombres del encabezado de la biblioteca de tipos mediante una calificación explícita mediante el nombre del espacio de nombres . O bien, puede incluir la siguiente instrucción:

```
using namespace MyLib;
```

inmediatamente después de **la #import** en el código fuente.

El espacio de nombres se puede suprimir mediante el atributo no_namespace) de la **#import** directiva . Sin embargo, la supresión del espacio de nombres puede dar lugar a conflictos de nombres. El nombre del espacio de nombres también se puede cambiar mediante rename_namespace atributo .

El compilador proporciona la ruta de acceso completa a cualquier dependencia de biblioteca de tipos que requiera la biblioteca de tipos que está procesando actualmente. La ruta de acceso se escribe, en forma de comentarios, en el encabezado de la biblioteca de tipos (.TLH) que el compilador genera para cada biblioteca de tipos procesada.

Si una biblioteca de tipos contiene referencias a tipos definidos en otras bibliotecas de tipos, el archivo .TLH incluirá comentarios del siguiente tipo:

```
//
// Cross-referenced type libraries:
//
//  #import "c:\path\typelib0.tlb"
//
```

El nombre de archivo real **del #import** comentario es la ruta de acceso completa de la biblioteca de tipos a la que se hace referencia, tal y como se almacena en el Registro. Si se producen errores causados por definiciones de tipo que faltan, compruebe los comentarios en la parte principal de . TLH para ver qué bibliotecas de tipos dependientes deben importarse primero. Los errores probables son errores de sintaxis (por ejemplo, C2143, C2146, C2321), C2501 (faltan especificadores decl) o C2433 ("inline" no permitida en la declaración de datos) mientras se compila el archivo .TLI.

Para resolver errores de dependencia, determine cuál de los comentarios de dependencia no se proporciona de otro modo para los encabezados del sistema y, a continuación, proporcione una directiva **#import** en algún momento antes de la directiva **#import** de la biblioteca de tipos dependientes.

#import atributos

#import puede incluir opcionalmente uno o varios atributos. Estos atributos indican al compilador que modifique el contenido de los encabezados de la biblioteca de tipos. Se puede usar un símbolo de barra diagonal inversa ( ) para incluir líneas **** adicionales en una sola **#import** instrucción. Por ejemplo:

```
#import "test.lib" no_namespace \
   rename("OldName", "NewName")
```

Para obtener más información, vea #import atributos.

Específicos de C++: END

Vea también

Directivas de preprocesador

Compatibilidad con COM del compilador

#import atributos (C++)

11/08/2021 • 2 minutes to read

Proporciona vínculos a atributos usados con la `#import` directiva .

Específicos de Microsoft

Los atributos siguientes están disponibles para la `#import` directiva .

ATRIBUTO	DESCRIPCIÓN
auto_rename	Cambia de nombre las palabras reservadas de C++ al anexar dos caracteres de subrayado (__) al nombre de la variable para resolver posibles conflictos de nombre.
auto_search	Especifica que, cuando se hace referencia a una biblioteca de tipos mediante #import y ella misma hace referencia a otra biblioteca de tipos, el compilador puede realizar un #import implícito para la otra biblioteca de tipos.
embedded_idl	Especifica que la biblioteca de tipos se escriba en el archivo .tlh, conservando el código generado por el atributo.
Excluir	Excluye elementos de los archivos de encabezado de la biblioteca de tipos que se generan.
high_method_prefix	Especifica un prefijo que se utilizará para designar propiedades y métodos de alto nivel.
high_property_prefixes	Especifica los prefijos alternativos para tres métodos de propiedad.
implementation_only	Suprime la generación de archivos de encabezado .tlh (el archivo de encabezado principal).
include()	Deshabilita la exclusión automática.
inject_statement	Inserta el argumento como texto original en el encabezado de la biblioteca de tipos.
named_guids	Indica al compilador que defina e inicialice variables GUID en el estilo anterior, con el formato `LIBID_MyLib` `CLSID_MyCoClass` , , y `IID_MyInterface` `DIID_MyDispInterface` .
no_auto_exclude	Deshabilita la exclusión automática.
no_dual_interfaces	Cambia la manera en que el compilador genera las funciones de contenedor para los métodos de interfaz dual.
no_implementation	Suprime la generación del encabezado .tli, que contiene las implementaciones de las funciones miembro de contenedor.

ATRIBUTO	DESCRIPCIÓN
no_namespace	Especifica que el compilador no genera el espacio de nombres.
no_registry	Indica al compilador que no busque en el Registro las bibliotecas de tipos.
no_search_namespace	Tiene la misma funcionalidad que el atributo no_namespace, pero se usa en las bibliotecas de tipos que usa la directiva #import con el atributo auto_search tipo.
no_smart_pointers	Suprime la creación de punteros inteligentes para todas las interfaces en la biblioteca de tipos.
raw_dispinterfaces	Indica al compilador que genere funciones contenedoras de bajo nivel para métodos y propiedades dispinterface que llaman `IDispatch::Invoke` y devuelven el código de error HRESULT.
raw_interfaces_only	Suprime la generación de funciones contenedoras de control de errores y declaraciones de propiedades que usan esas funciones contenedoras.
raw_method_prefix	Especifica otro prefijo para evitar conflictos de nombres.
raw_native_types	Deshabilita el uso de las clases de soporte de COM en las funciones de contenedor de alto nivel y, en su lugar, fuerza el uso de tipos de datos de bajo nivel.
raw_property_prefixes	Especifica los prefijos alternativos para tres métodos de propiedad.
rename	Resuelve problemas del conflicto de nombres.
rename_namespace	Cambia el espacio de nombres que incluye el contenido de la biblioteca de tipos.
rename_search_namespace	Tiene la misma funcionalidad que el atributo rename_namespace, pero se usa en bibliotecas de tipos que usa la directiva #import con el atributo auto_search tipo.
tlbid	Permite cargar bibliotecas distintas de la biblioteca de tipos primaria.

FIN de Específicos de Microsoft

Vea también

#import directiva

auto_rename de importación

12/08/2021 • 2 minutes to read

Específicos de C++

Cambia de nombre las palabras reservadas de C++ al anexar dos caracteres de subrayado (__) al nombre de la variable para resolver posibles conflictos de nombre.

Sintaxis

```
#import biblioteca de tipos auto_rename
```

Comentarios

Este atributo se usa al importar una biblioteca de tipos que utiliza una o más palabras reservadas de C++ (palabras clave o macros) como nombres de variable.

Específicos de C++: END

Consulte también

#import atributos
#import directiva

auto_search atributo de importación

17/08/2021 • 2 minutes to read

Específicos de C++

Especifica que, cuando se hace referencia a una biblioteca de tipos con y hace referencia a otra biblioteca de tipos, el compilador puede hacer un implícito `#import` para la otra biblioteca de `#import` tipos.

Sintaxis

```
#import biblioteca de tipos auto_search
```

Comentarios

Específicos de C++: END

Vea también

#import atributos
#import directiva

embedded_idl de importación

17/08/2021 • 2 minutes to read

Específicos de C++

Especifica si la biblioteca de tipos se escribe en el `.tlh` archivo con el código generado por atributos conservado.

Sintaxis

```
#import biblioteca de tipos embedded_idl [ ( { "emitidl" | "no_emitidl" } ) ]
```

Parámetros

"emitidl"
La información de tipo *importada de la biblioteca de* tipos está presente en el IDL generado para el proyecto con atributos. Este comportamiento es el predeterminado y está en vigor si no se especifica un parámetro para `embedded_idl` .

"no_emitidl"
La información de tipo *importada de la biblioteca de* tipos no está presente en el IDL generado para el proyecto con atributos.

Ejemplo

```
// import_embedded_idl.cpp
// compile with: /LD
#include <windows.h>
[module(name="MyLib2")];
#import "\school\bin\importlib.tlb" embedded_idl("no_emitidl")
```

Específicos de C++: END

Vea también

#import atributos
#import directiva

excluir atributo de importación

17/08/2021 • 2 minutes to read

Específicos de C++

Excluye elementos de los archivos de encabezado de la biblioteca de tipos que se generan.

Sintaxis

```
#import la biblioteca de tipos exclude( "Name1" [ , "Name2" ... ] )
```

Parámetros

Nombre1
Primer elemento que se excluirá.

Nombre2
(Opcional) Segundo y posteriores elementos que se excluirán, si es necesario.

Comentarios

Las bibliotecas de tipos pueden incluir definiciones de elementos definidos en encabezados de sistema u otras bibliotecas de tipos. Este atributo puede tomar cualquier número de argumentos, donde cada uno es un elemento de biblioteca de tipos de nivel superior que se va a excluir.

Específicos de C++: END

Vea también

#import atributos
#import directiva

high_method_prefix atributo de importación

16/08/2021 • 2 minutes to read

Específicos de C++

Especifica un prefijo que se utilizará para designar propiedades y métodos de alto nivel.

Sintaxis

```
#import biblioteca de tipos high_method_prefix( "Prefijo" )
```

Parámetros

Prefijo
Prefijo que se va a utilizar.

Comentarios

De forma predeterminada, las propiedades y los métodos de control de errores de alto nivel se exponen mediante funciones miembro denominadas sin prefijo. Los nombres son de la biblioteca de tipos.

Específicos de C++: END

Vea también

#import atributos
#import directiva

high_property_prefixes atributo de importación

16/08/2021 • 2 minutes to read

Específicos de C++

Especifica los prefijos alternativos para tres métodos de propiedad.

Sintaxis

```
#import biblioteca de tipos high_property_prefixes( "GetPrefix" , "PutPrefix" , "PutRefPrefix" )
```

Parámetros

GetPrefix
Prefijo que se va a usar para los `propget` métodos.

PutPrefix
Prefijo que se va a usar para los `propput` métodos.

PutRefPrefix
Prefijo que se va a usar para los `propputref` métodos.

Comentarios

De forma predeterminada, los métodos , y de control de errores de alto nivel se exponen mediante funciones miembro denominadas con `propget` `propput` `propputref` prefijos `Get` , y , `Put` `PutRef` respectivamente.

Específicos de C++: END

Vea también

#import atributos
#import directiva

implementation_only atributo import

16/08/2021 • 2 minutes to read

Específicos de C++

Suprime la generación del `.tlh` archivo de encabezado principal de la biblioteca de tipos.

Sintaxis

```
#import biblioteca de tipos implementation_only
```

Comentarios

Este archivo contiene todas las declaraciones utilizadas para exponer el contenido de la biblioteca de tipos. El `.tli` archivo de encabezado, con las implementaciones de las funciones miembro contenedoras, se generará e incluirá en la compilación.

Cuando se especifica este atributo, el contenido del encabezado se encuentra en el mismo espacio de nombres que el que `.tli` se usa normalmente en el `.tlh` encabezado. Además, las funciones miembro no se declaran como alineadas.

El **implementation_only** está pensado para su uso junto con el atributo no_implementation como una manera de mantener las implementaciones fuera del archivo de encabezado precompilado (PCH). Una instrucción `#import` con el atributo `no_implementation` se coloca en la región de origen usada para crear el PCH. Varios archivos de código fuente utilizan el PCH resultante. A continuación, se usa una instrucción implementation_only atributo fuera `#import` de la región PCH. Debe usar esta instrucción solo una vez en uno de los archivos de origen. Genera todas las funciones miembro contenedoras necesarias sin recompilación adicional para cada archivo de origen.

> **NOTE**
> El **implementation_only** en una instrucción debe usarse junto con otra instrucción, de la misma biblioteca `#import` de `#import` tipos, con el atributo `no_implementation` . De lo contrario, se generan errores del compilador. Esto se debe a que las definiciones de clase contenedoras generadas por la instrucción con el atributo son necesarias para compilar las implementaciones generadas por `#import` `no_implementation` el atributo **implementation_only** contenedor.

Específicos de C++: END

Vea también

#import atributos
#import directiva

Atributo de importación include()

12/08/2021 • 2 minutes to read

Específicos de C++

Deshabilita la exclusión automática.

Sintaxis

```
#import la biblioteca de tipos include( "Name1" [,"Name2" ... ] )
```

Parámetros

Nombre1
Primer elemento que se incluirá forzosamente.

Nombre2
Segundo elemento que se incluirá forzosamente (si es necesario).

Comentarios

Las bibliotecas de tipos pueden incluir definiciones de elementos definidos en encabezados de sistema u otras bibliotecas de tipos. `#import` intenta evitar varios errores de definición mediante la exclusión automática de dichos elementos. Si algunos elementos no se deben excluir automáticamente, es posible que vea Advertencia del compilador (nivel 3) C4192. Puede usar este atributo para deshabilitar la exclusión automática. Este atributo puede tomar cualquier número de argumentos, uno para cada nombre de un elemento de biblioteca de tipos que se va a incluir.

Específicos de C++: END

Consulte también

#import atributos
#import directiva

inject_statement atributo import

14/08/2021 • 2 minutes to read

Específicos de C++

Inserta el argumento como texto original en el encabezado de la biblioteca de tipos.

Sintaxis

```
#import biblioteca de tipos inject_statement( "source-text" )
```

Parámetros

source-text
Texto original que se inserta en el archivo de encabezado de la biblioteca de tipos.

Comentarios

El texto se coloca al principio de la declaración de espacio de nombres que encapsula el contenido *de la biblioteca de* tipos en el archivo de encabezado.

Específicos de C++: END

Vea también

#import atributos
#import directiva

named_guids atributo import

13/08/2021 • 2 minutes to read

Específicos de C++

Indica al compilador que defina e inicialice variables GUID en el estilo anterior, con el formato `LIBID_MyLib` `CLSID_MyCoClass` , , y `IID_MyInterface` `DIID_MyDispInterface` .

Sintaxis

```
#import biblioteca de tipos named_guids
```

Específicos de C++: END

Vea también

#import atributos
#import directiva

no_auto_exclude de importación

16/08/2021 • 2 minutes to read

Específicos de C++

Deshabilita la exclusión automática.

Sintaxis

```
#import biblioteca de tipos no_auto_exclude
```

Comentarios

Las bibliotecas de tipos pueden incluir definiciones de elementos definidos en encabezados de sistema u otras bibliotecas de tipos. `#import` intenta evitar varios errores de definición mediante la exclusión automática de dichos elementos. Hace que se emita advertencia del compilador (nivel 3) C4192 para cada elemento que se va a excluir. Puede deshabilitar la exclusión automática mediante este atributo.

Específicos de C++: END

Vea también

#import atributos
#import directiva

no_dual_interfaces atributo import

14/08/2021 • 2 minutes to read

Específicos de C++

Cambia la manera en que el compilador genera las funciones de contenedor para los métodos de interfaz dual.

Sintaxis

```
#import biblioteca de tipos no_dual_interfaces
```

Comentarios

Normalmente, el contenedor llama al método a través de la tabla de funciones virtuales para la interfaz . Con **no_dual_interfaces**, el contenedor llama en su lugar `IDispatch::Invoke` a para invocar el método .

Específicos de C++: END

Vea también

#import atributos
#import directiva

no_implementation atributo de importación

13/08/2021 • 2 minutes to read

Específicos de C++

Suprime la generación del `.tli` encabezado , que contiene las implementaciones de las funciones miembro contenedoras.

Sintaxis

```
#import biblioteca de tipos no_implementation
```

Comentarios

Si se especifica este atributo, el encabezado , con las declaraciones para exponer elementos de biblioteca de tipos, se generará sin una instrucción para incluir `.tlh` `#include` el archivo de `.tli` encabezado.

Este atributo se usa junto con implementation_only.

Específicos de C++: END

Vea también

#import atributos
#import directiva

no_namespace atributo import

11/08/2021 • 2 minutes to read

Específicos de C++

Especifica que el compilador no genera un nombre de espacio de nombres.

Sintaxis

```
#import biblioteca de tipos no_namespace
```

Comentarios

El contenido de la biblioteca de tipos en el archivo de encabezado `#import` se define normalmente en un espacio de nombres. El nombre del espacio de nombres se especifica en la `library` instrucción del archivo IDL original. Si se **no_namespace** atributo , el compilador no genera este espacio de nombres.

Si desea usar un nombre de espacio de nombres diferente, use el atributo rename_namespace en su lugar.

Específicos de C++: END

Consulte también

#import atributos
#import directiva

no_registry de importación

12/08/2021 • 2 minutes to read

no_registry indica al compilador que no busque en el Registro bibliotecas de tipos importadas con `#import` .

Sintaxis

```
#import biblioteca de tipos no_registry
```

Parámetros

type-library
Una biblioteca de tipos.

Comentarios

Si no se encuentra una biblioteca de tipos a la que se hace referencia en los directorios de include, se produce un error en la compilación incluso si la biblioteca de tipos está en el Registro. **no_registry** propaga a otras bibliotecas de tipos importadas implícitamente con `auto_search` .

El compilador nunca busca en el Registro bibliotecas de tipos especificadas por el nombre de archivo y que se pasan directamente a `#import` .

Cuando se especifica , las directivas adicionales se generan mediante el no_registry `auto_search` `#import` del valor `#import` inicial. Si la directiva `#import` inicial se **no_registry**, también se genera `auto_search` un `#import` **no_registry**.

no_registry es útil si desea importar bibliotecas de tipos a las que se hace referencia cruzada. Impide que el compilador encuentre una versión anterior del archivo en el Registro. **no_registry** también es útil si la biblioteca de tipos no está registrada.

Consulte también

#import atributos
#import directiva

no_search_namespace atributo import

17/08/2021 • 2 minutes to read

Específicos de C++

Tiene la misma funcionalidad que el atributo no_namespace, pero se usa en bibliotecas de tipos donde se usa la directiva con el `#import` auto_search atributo .

Sintaxis

```
#import biblioteca de tipos no_search_namespace
```

Específicos de C++: END

Vea también

#import atributos
#import directiva

no_smart_pointers atributo import

16/08/2021 • 2 minutes to read

Específicos de C++

Suprime la creación de punteros inteligentes para todas las interfaces en la biblioteca de tipos.

Sintaxis

```
#import biblioteca de tipos no_smart_pointers
```

Comentarios

De forma predeterminada, cuando se utiliza `#import`, se obtiene una declaración de puntero inteligente para todas las interfaces de la biblioteca de tipos. Estos punteros inteligentes son de tipo _com_ptr_t.

Específicos de C++: END

Vea también

#import atributos
#import directiva

raw_dispinterfaces de importación

16/08/2021 • 2 minutes to read

Específicos de C++

Indica al compilador que genere funciones contenedoras de bajo nivel para los métodos dispinterface y para las propiedades que llaman y `IDispatch::Invoke` devuelven el código de error HRESULT.

Sintaxis

```
#import biblioteca de tipos raw_dispinterfaces
```

Comentarios

Si no se especifica este atributo, solo se generan contenedores de alto nivel, que inician excepciones de C++ en caso de error.

Específicos de C++: END

Vea también

#import atributos
#import directiva

raw_interfaces_only atributo import

12/08/2021 • 2 minutes to read

Específicos de C++

Suprime la generación de funciones contenedoras de control de errores y declaraciones de propiedades que usan esas funciones contenedoras.

Sintaxis

```
#import biblioteca de tipos raw_interfaces_only
```

Comentarios

El **raw_interfaces_only** atributo también hace que se quite el prefijo predeterminado utilizado para asignar un nombre a las funciones que no son de propiedad. Normalmente, el prefijo es `raw_` . Si se especifica este atributo, los nombres de función se toman directamente de la biblioteca de tipos.

Este atributo permite exponer solo el contenido de bajo nivel de la biblioteca de tipos.

Específicos de C++: END

Consulte también

#import atributos
#import directiva

raw_method_prefix

16/08/2021 • 2 minutes to read

Específicos de C++

Especifica otro prefijo para evitar conflictos de nombres.

Sintaxis

```
#import biblioteca de tipos raw_method_prefix( "Prefijo" )
```

Parámetros

Prefijo
El prefijo que se va a usar.

Comentarios

Las propiedades y los métodos de bajo nivel se exponen mediante funciones miembro denominadas mediante un prefijo predeterminado de **raw_**, para evitar conflictos de nombres con las funciones miembro de control de errores de alto nivel.

> **NOTE**
> Los efectos del atributo **raw_method_prefix** no se modifican por la presencia del raw_interfaces_only atributo. El **raw_method_prefix** siempre tiene prioridad sobre `raw_interfaces_only` al especificar un prefijo. Si ambos atributos se usan en la misma instrucción, se usa el prefijo especificado `#import` **por raw_method_prefix** atributo .

Específicos de C++: END

Vea también

#import atributos
#import directiva

raw_native_types atributo import

11/08/2021 • 2 minutes to read

Específicos de C++

Deshabilita el uso de clases de compatibilidad COM en las funciones contenedoras de alto nivel y fuerza el uso de tipos de datos de bajo nivel en su lugar.

Sintaxis

```
#import biblioteca de tipos raw_native_types
```

Comentarios

De forma predeterminada, los métodos de control de errores de alto nivel usan las clases de compatibilidad COM _bstr_t y _variant_t en lugar de los tipos de datos y y los punteros de interfaz COM sin `BSTR` `VARIANT` formato. Estas clases encapsulan los detalles de asignación y desasignación del almacenamiento de memoria para estos tipos de datos, y simplifican considerablemente la conversión de tipos y las operaciones de conversión.

Específicos de C++: END

Consulte también

#import atributos
#import directiva

raw_property_prefixes atributo import

16/08/2021 • 2 minutes to read

Específicos de C++

Especifica los prefijos alternativos para tres métodos de propiedad.

Sintaxis

```
#import biblioteca de tipos raw_property_prefixes( "GetPrefix" , "PutPrefix" , "PutRefPrefix" )
```

Parámetros

GetPrefix
Prefijo que se usará para los `propget` métodos.

PutPrefix
Prefijo que se usará para los `propput` métodos.

PutRefPrefix
Prefijo que se usará para los `propputref` métodos.

Comentarios

De forma predeterminada, los métodos , y de bajo nivel se exponen mediante funciones miembro denominadas con `propget` `propput` `propputref` prefijos `get_` de , y `put_` , `putref_` respectivamente. Estos prefijos son compatibles con los nombres que se usan en los archivos de encabezado generados por MIDL.

Específicos de C++: END

Vea también

#import atributos
#import directiva

cambiar el nombre del atributo de importación

16/08/2021 • 2 minutes to read

Específicos de C++

Resuelve problemas del conflicto de nombres.

Sintaxis

```
#import de la biblioteca de tipos rename( "OldName" , "NewName" )
```

Parámetros

OldName
Nombre anterior en la biblioteca de tipos.

Newname
Nombre usado en lugar del nombre anterior.

Comentarios

Cuando se especifica el atributo **rename,** el compilador reemplaza todas las apariciones de *OldName* en la biblioteca de tipos por el *newName* proporcionado por el usuario en los archivos de encabezado resultantes.

El **atributo rename** se puede usar cuando un nombre de la biblioteca de tipos coincide con una definición de macro en los archivos de encabezado del sistema. Si no se resuelve esta situación, el compilador puede emitir varios errores de sintaxis, como error del compilador C2059 y error del compilador C2061.

> **NOTE**
> La sustitución se aplica a un nombre usado en la biblioteca de tipos, no a un nombre usado en el archivo de encabezado resultante.

Suponga, por ejemplo, que hay una propiedad denominada `MyParent` en una biblioteca de tipos y que se define una macro `GetMyParent` en un archivo de encabezado y se utiliza antes de `#import`. Puesto `GetMyParent` que es el nombre predeterminado de una función contenedora para la propiedad de control de `get` errores, se producirá una colisión de nombres. Para solucionar el problema, utilice el siguiente atributo en la instrucción `#import`:

```
#import MyTypeLib.tlb rename("MyParent","MyParentX")
```

que cambia el nombre `MyParent` en la biblioteca de tipos. Un intento de cambiar el nombre del contenedor `GetMyParent` producirá un error:

```
#import MyTypeLib.tlb rename("GetMyParent","GetMyParentX")
```

Se debe a que el nombre solo aparece en el archivo de encabezado `GetMyParent` de la biblioteca de tipos resultante.

Específicos de C++: END

Vea también

#import atributos
#import directiva

rename_namespace atributo import

13/08/2021 • 2 minutes to read

Específicos de C++

Cambia el espacio de nombres que incluye el contenido de la biblioteca de tipos.

Sintaxis

```
#import biblioteca de tipos rename_namespace( "NewName" )
```

Parámetros

Newname
Nuevo nombre del espacio de nombres.

Comentarios

El **rename_namespace** toma un único argumento, *NewName*, que especifica el nuevo nombre para el espacio de nombres.

Para quitar el espacio de nombres, use el atributo no_namespace en su lugar.

Específicos de C++: END

Vea también

#import atributos
#import directiva

rename_search_namespace de importación

17/08/2021 • 2 minutes to read

Específicos de C++

Tiene la misma funcionalidad que el atributo rename_namespace, pero se usa en bibliotecas de tipos donde se usa la directiva junto con el `#import` atributo auto_search.

Sintaxis

```
#import biblioteca de tipos rename_search_namespace( "NewName" )
```

Parámetros

Newname
Nuevo nombre del espacio de nombres.

Comentarios

Específicos de C++: END

Vea también

#import atributos
#import directiva

Atributo de importación tlbid

16/08/2021 • 2 minutes to read

Específicos de C++

Permite cargar bibliotecas distintas de la biblioteca de tipos primaria.

Sintaxis

```
#import type-library-dll tlbid( number )
```

Parámetros

Número
Número de la biblioteca de tipos en *type-library-dll*.

Comentarios

Si hay varias bibliotecas de tipos integradas en un único archivo DLL, es posible cargar bibliotecas distintas de la biblioteca de tipos principal mediante **tlbid**.

Por ejemplo:

```
#import <MyResource.dll> tlbid(2)
```

equivale a:

```
LoadTypeLib("MyResource.dll\\2");
```

Específicos de C++: END

Vea también

#import atributos
#import directiva

`#include` (C/C++)

11/08/2021 • 5 minutes to read

Indica al preprocesador que incluya el contenido de un archivo especificado en el punto donde aparece la directiva .

Sintaxis

```
#include " path-spec "
#include < path-spec >
```

Comentarios

Puede organizar las definiciones de constantes y macros en archivos de incluir *(también* conocidos como archivos de *encabezado)* y, a continuación, usar directivas para agregarlas `#include` a cualquier archivo de origen. Los archivos de inclusión son también útiles para incorporar declaraciones de variables externas y tipos de datos complejos. Los tipos solo se pueden definir y denominar una sola vez en un archivo de inclusión creado para ese propósito.

La *especificación de ruta* de acceso es un nombre de archivo que, opcionalmente, puede ir precedido de una especificación de directorio. El nombre de archivo debe designar un archivo existente. La sintaxis de *path-spec* depende del sistema operativo en el que se compila el programa.

Para obtener información sobre cómo hacer referencia a ensamblados en una aplicación de C++ compilada mediante `/clr` , vea `#using` la directiva.

Ambas formas de sintaxis `#include` hacen que la directiva se reemplazó por todo el contenido del archivo especificado. La diferencia entre las dos formas es el orden de las rutas de acceso que busca el preprocesador cuando la ruta de acceso está incompletamente especificada. En la siguiente tabla se muestran las diferencias entre ambos formatos de sintaxis.

FORMATO DE SINTAXIS	ACCIÓN
Formato con comillas	El preprocesador realiza la búsqueda de archivos de inclusión en este orden: 1) En el mismo directorio que el archivo que contiene la `#include` instrucción . 2) En los directorios de los archivos de incluyen abiertos actualmente, en el orden inverso en el que se han abierto. La búsqueda comienza en el directorio del archivo de inclusión principal y continúa hacia arriba por los directorios de cualquier archivo de inclusión primario principal. 3) A lo largo de la ruta de acceso especificada por cada opción `/I` del compilador. 4) A lo largo de las rutas de acceso especificadas por la `INCLUDE` variable de entorno.

FORMATO DE SINTAXIS	ACCIÓN
Formato con corchetes angulares	El preprocesador realiza la búsqueda de archivos de inclusión en este orden: 1) A lo largo de la ruta de acceso especificada por cada opción `/I` del compilador. 2) Cuando se produce la compilación en la línea de comandos, a lo largo de las rutas de acceso especificadas por la `INCLUDE` variable de entorno.

El preprocesador detiene la búsqueda en cuanto encuentra un archivo con el nombre especificado. Si incluye una especificación de ruta de acceso completa e inequívoca para el archivo de incluir entre comillas dobles ( ), el preprocesador busca solo esa especificación de ruta de acceso y omite los `" "` directorios estándar.

Si el nombre de archivo que se incluye entre comillas dobles es una especificación de ruta de acceso incompleta, el preprocesador busca primero en el directorio del archivo primario. Un archivo primario es el archivo que contiene la `#include` directiva . Por ejemplo, si incluye un archivo denominado *file2* en un archivo denominado *file1*, *file1* es el archivo primario.

Los archivos de incluir se *pueden anidar:* una directiva puede aparecer en un archivo denominado `#include` por otra `#include` directiva. Por ejemplo, *file2* podría incluir *file3*. En este caso, *file1* seguiría siendo el elemento primario de *file2*, pero sería el elemento primario principal de *file3*.

Cuando los archivos include están anidados y cuando la compilación se produce en la línea de comandos, la búsqueda de directorios comienza en el directorio del archivo primario. A continuación, continúa por los directorios de los archivos primarios primarios. Es decir, la búsqueda comienza en relación con el directorio que contiene el origen que se procesa actualmente. Si no se encuentra el archivo, la búsqueda se mueve a los directorios especificados por la opción del compilador `/I` (Directorios de incluir adicionales). Por último, se buscan los directorios especificados por la `INCLUDE` variable de entorno.

Dentro del Visual Studio de desarrollo, se `INCLUDE` omite la variable de entorno. En su lugar, se usan los valores especificados en las propiedades del proyecto para los directorios de include. Para obtener más información sobre cómo establecer los directorios de Visual Studio, vea Directorios de incluir y directorios de incluir adicionales.

Este ejemplo muestra la inclusión de archivo mediante corchetes angulares:

```
#include <stdio.h>
```

En el ejemplo se agrega el contenido del archivo denominado `stdio.h` al programa de origen. Los corchetes angulares hacen que el preprocesador busque en los directorios especificados por la variable de entorno para , después de buscar en los directorios especificados por la opción `INCLUDE` `stdio.h` del `/I` compilador.

En el ejemplo siguiente se muestra la inclusión de archivo mediante el formato con comillas:

```
#include "defs.h"
```

En el ejemplo se agrega el contenido del archivo especificado por `defs.h` al programa de origen. Las comillas indican que el preprocesador busca primero en el directorio que contiene el archivo de código fuente primario.

El anidamiento de archivos de inclusión puede continuar hasta 10 niveles. Cuando finaliza el procesamiento del anidado, el preprocesador continúa insertando el archivo de incluir primario en el `#include` archivo de origen original.

Específico de Microsoft

Para buscar los archivos de origen que se incluirán, el preprocesador busca primero en los directorios especificados por la `/I` opción del compilador. Si la opción no está presente o si se produce un error, el preprocesador usa la variable de entorno para buscar los archivos de incluir entre `/I` `INCLUDE` corchetes angulares. La `INCLUDE` variable de entorno y la opción del compilador pueden contener varias `/I` rutas de acceso, separadas por punto y coma (`;`). Si aparece más de un directorio como parte de la opción o dentro de la variable de entorno, el preprocesador los busca en el orden en `/I` `INCLUDE` que aparecen.

Por ejemplo, el comando

```
CL /ID:\msvc\include myprog.c
```

hace que el preprocesador busque en el directorio `D:\msvc\include\` archivos de incluyeción, como `stdio.h` .
Los comandos

```
SET INCLUDE=D:\msvc\include
CL myprog.c
```

producen el mismo efecto. Si se produce un error en ambos conjuntos de búsquedas, se genera un error grave del compilador.

Si el nombre de archivo está totalmente especificado para un archivo de incluir que tiene una ruta de acceso que incluye dos puntos (por ejemplo,), el preprocesador sigue `F:\MSVC\SPECIAL\INCL\TEST.H` la ruta de acceso.

En el caso de los archivos include que se especifican como , la búsqueda de directorios comienza en el directorio del archivo primario y, a continuación, continúa por los directorios de `#include "path-spec"` los archivos primarios primarios. Es decir, la búsqueda comienza en relación con el directorio que contiene el archivo de origen que se está procesando. Si no hay ningún archivo primario general y el archivo todavía no se encuentra, la búsqueda continúa como si el nombre de archivo estuviera entre corchetes angulares.

FIN de Específicos de Microsoft

Consulte también

Directivas de preprocesador
`/I` (Directorios de incluir adicionales)

#line directiva (C/C++)

15/08/2021 • 2 minutes to read

La **#line** indica al preprocesador que establezca los valores notificados del compilador para el número de línea y el nombre de archivo en un número de línea y un nombre de archivo determinados.

Sintaxis

```
** #line **digit-sequence ["filename"]
```

Comentarios

El compilador utiliza el número de línea y el nombre de archivo opcional para hacer referencia a los errores que encuentra durante la compilación. El número de línea suele referirse a la línea de entrada actual, y el nombre de archivo hace referencia al archivo de entrada actual. El número de línea se incrementa una vez que se procesa cada línea.

El *valor de secuencia de* dígitos puede ser cualquier constante de entero. El reemplazo de macros se puede usar en los tokens de preprocesamiento, pero el resultado debe evaluarse con la sintaxis correcta. El *nombre de* archivo puede ser cualquier combinación de caracteres y debe incluirse entre comillas dobles (`" "`). Si *se omite filename,* el nombre de archivo anterior permanece sin cambios.

Puede modificar el número de línea de origen y el nombre de archivo escribiendo una `#line` directiva . La `#line` directiva establece el valor de la línea que sigue inmediatamente a la directiva en el archivo de origen. El traductor usa el número de línea y el nombre de archivo para determinar los valores de las macros predefinidas `__FILE__` y `__LINE__` . Estas macros se pueden utilizar para insertar mensajes de error autodescriptivos en el texto del programa. Para obtener más información sobre estas macros predefinidas, vea Macros predefinidas.

La `__FILE__` macro se expande a una cadena cuyo contenido es el nombre de archivo, entre comillas dobles (`" "`).

Si se cambia el número de línea y el nombre de archivo, el compilador omite los valores anteriores y continúa el procesamiento con los nuevos valores. Los **generadores #line** suelen usar la directiva de #line programa. Se usa para hacer que los mensajes de error hacen referencia al archivo de origen original, en lugar de al programa generado.

Ejemplo

En los ejemplos siguientes se `#line` muestran las macros y `__LINE__` `__FILE__` .

En el primer ejemplo, el número de línea se establece en 10, después en 20 y el nombre de archivo se cambia a *hello.cpp*.

```
// line_directive.cpp
// Compile by using: cl /W4 /EHsc line_directive.cpp
#include <stdio.h>

int main()
{
    printf( "This code is on line %d, in file %s\n", __LINE__, __FILE__ );
#line 10
    printf( "This code is on line %d, in file %s\n", __LINE__, __FILE__ );
#line 20 "hello.cpp"
    printf( "This code is on line %d, in file %s\n", __LINE__, __FILE__ );
    printf( "This code is on line %d, in file %s\n", __LINE__, __FILE__ );
}
```

```
This code is on line 7, in file line_directive.cpp
This code is on line 10, in file line_directive.cpp
This code is on line 20, in file hello.cpp
This code is on line 21, in file hello.cpp
```

En este ejemplo, la macro usa las macros predefinidas y para imprimir un mensaje de error sobre el archivo de origen si una aserción determinada `ASSERT` `__LINE__` no es `__FILE__` verdadera.

```
#define ASSERT(cond) if( !(cond) )\
{printf( "assertion error line %d, file(%s)\n", \
__LINE__, __FILE__ );}
```

Vea también

Directivas de preprocesador

NULL (Directiva)

13/08/2021 • 2 minutes to read

La directiva de preprocesador NULL es un signo de número único (#) solo en una línea. No tiene efecto.

Sintaxis

```
#
```

Vea también

Directivas de preprocesador

#undef directiva (C/C++)

12/08/2021 • 2 minutes to read

Quita (anula la definición de) un nombre creado previamente con `#define`.

Sintaxis

> **#undef** *identificador*

Comentarios

La **#undef** directiva quita la definición actual del *identificador*. Por lo tanto, el preprocesador omite las apariciones posteriores del identificador. Para quitar una definición de macro **#undef**, dé solo el identificador *de* macro , no una lista de parámetros.

También puede aplicar la directiva **#undef** a un identificador que no tenga ninguna definición anterior. De este modo se garantiza que el identificador no esté definido. El reemplazo de macro no se realiza **dentro #undef** instrucciones.

La **#undef** directiva se empareja normalmente con una directiva para crear una región en un programa de origen en el que un identificador `#define` tiene un significado especial. Por ejemplo, una función específica del programa de origen puede utilizar constantes de manifiesto para definir valores específicos del entorno que no afecten al resto del programa. La **#undef** directiva también funciona con la `#if` directiva para controlar la compilación condicional del programa de origen. Para obtener más información, vea The #if, #elif, #else, and #endif directives.

En el ejemplo siguiente, la **directiva #undef** elimina las definiciones de una constante simbólica y una macro. Observe que solo se proporciona el identificador de la macro.

```
#define WIDTH 80
#define ADD( X, Y ) ((X) + (Y))
.
.
.
#undef WIDTH
#undef ADD
```

Específicos de Microsoft

Las macros pueden no estar definidas desde la línea de comandos mediante la `/U` opción , seguida de los nombres de macro que se va a no definir. El efecto de emitir este comando es equivalente a una secuencia de `#undef` *instrucciones de nombre* de macro al principio del archivo.

FIN de Específicos de Microsoft

Vea también

Directivas de preprocesador

#using directiva (C++/CLI)

11/08/2021 • 3 minutes to read

Importa metadatos en un programa compilado con /clr.

Sintaxis

```
** #using ** file [ as_friend ]
```

Parámetros

Archivo
Un archivo de lenguaje intermedio de Microsoft (MSIL), `.dll` `.exe` , o `.netmodule` `.obj` . Por ejemplo,

```
#using <MyComponent.dll>
```

`as_friend`
Especifica que todos los tipos del *archivo son* accesibles. Para obtener más información, vea Ensamblados de confianza (C++).

Comentarios

file puede ser un archivo de lenguaje intermedio de Microsoft (MSIL) que se importa para sus datos administrados y construcciones administradas. Si un archivo DLL contiene un manifiesto de ensamblado, se importan todos los archivos DLL a los que se hace referencia en el manifiesto. El ensamblado que va a compilar mostrará el archivo *en* los metadatos como referencia de ensamblado.

Es *posible* que el archivo no contenga un ensamblado *(el* archivo es un módulo) y no pretende usar la información de tipo del módulo en la aplicación actual (ensamblado). Puede indicar que el módulo forma parte del ensamblado mediante /ASSEMBLYMODULE. Los tipos del módulo estarán disponibles entonces para cualquier aplicación que hiciera referencia al ensamblado.

Una alternativa a usar `#using` es la opción del compilador /FU.

.exe ensamblados pasados a se deben compilar mediante uno de los compiladores de `#using` .NET Visual Studio (Visual Basic o Visual C#, por ejemplo). Al intentar importar metadatos desde un ensamblado .exe compilado con `/clr` , se producirá una excepción de carga de archivos.

> **NOTE**
> Un componente al que se hace referencia con se puede ejecutar con una versión diferente del archivo importado en tiempo de compilación, lo que hace que una aplicación cliente `#using` dé resultados inesperados.

Para que el compilador reconozca un tipo en un ensamblado (no un módulo), debe forzarse para resolver el tipo. Puede forzarla, por ejemplo, definiendo una instancia del tipo . Hay otras maneras de resolver nombres de tipo en un ensamblado para el compilador. Por ejemplo, si hereda de un tipo en un ensamblado, el compilador conoce el nombre del tipo.

Al importar metadatos creados a partir del código fuente que usaba , la semántica del subproceso `__declspec(thread)` no se conserva en los metadatos. Por ejemplo, una variable declarada con , compilada en un programa que se compila para .NET Framework Common Language Runtime y, a continuación, se importa a

través de , no tendrá `__declspec(thread)` `#using` semántica en la `__declspec(thread)` variable.

Todos los tipos importados (administrados y nativos) de un archivo al que hace referencia están disponibles, pero el compilador trata los tipos nativos como `#using` declaraciones, no definiciones.

mscorlib.dll se hace referencia automáticamente al compilar con `/clr` .

La variable de entorno LIBPATH especifica los directorios que se buscarán cuando el compilador resuelva los nombres de archivo pasados a `#using` .

El compilador busca referencias a lo largo de la ruta de acceso siguiente:

- Ruta de acceso especificada en la `#using` instrucción .
- El directorio actual.
- El directorio del sistema de .NET Framework.
- Directorios agregados con la `/AI` opción del compilador.
- Directorios de la variable de entorno LIBPATH.

Ejemplos

Puede compilar un ensamblado que haga referencia a un segundo ensamblado que a su vez haga referencia a un tercer ensamblado. Solo tiene que hacer referencia explícitamente al tercer ensamblado del primero si usa explícitamente uno de sus tipos.

```
// using_assembly_A.cpp
// compile with: /clr /LD
public ref class A {};
```

```
// using_assembly_B.cpp
// compile with: /clr /LD
#using "using_assembly_A.dll"
public ref class B {
public:
   void Test(A a) {}
   void Test() {}
};
```

En el ejemplo siguiente, el compilador no informa de un error sobre la referencia *ausing_assembly_A.dll*, porque el programa no usa ninguno de los tipos definidos *en using_assembly_A.cpp*.

```
// using_assembly_C.cpp
// compile with: /clr
#using "using_assembly_B.dll"
int main() {
   B b;
   b.Test();
}
```

Consulte también

Directivas de preprocesador

Operadores de preprocesador

13/08/2021 • 2 minutes to read

Se usan cuatro operadores específicos del preprocesador en el contexto de la `#define` directiva . Consulte la tabla siguiente para obtener un resumen de cada una de ellas. En las próximas tres secciones se explican los operadores de generación de cadenas, generación de caracteres y pegado de token. Para obtener información sobre `defined` el operador , vea The #if, #elif, #else y #endif directivas.

OPERATOR	ACCIÓN
Operador de conversión a cadenas (#)	Hace que el argumento real correspondiente se delimite con comillas dobles
Operador de generación de caracteres (#@)	Hace que el argumento correspondiente se entre comillas simples y se trate como un carácter (específico de Microsoft)
Operador de pegado de token (##)	Permite concatenar tokens utilizados como argumentos reales para formar otros tokens
Operador defined	Simplifica la escritura de expresiones compuestas en determinadas directivas de macro

Vea también

Directivas de preprocesador
Macros predefinidas
Referencia del preprocesador de c/c++

Operador de conversión a cadenas (#)

15/08/2021 • 2 minutes to read

El operador number-sign o "stringizing" ( ) convierte los parámetros de macro en literales de cadena sin # expandir la definición de parámetro. Solo se usa con macros que toman argumentos. Si precede un parámetro formal en la definición de macro, el argumento real pasado por la llamada de macro se pone entre comillas y se trata como un literal de cadena. El literal de cadena reemplaza cada aparición de una combinación de operador de generación de cadenas y parámetro formal dentro de la definición de macro.

> **NOTE**
> Ya no se admite la extensión de Microsoft C (versión 6.0 y anteriores) para el estándar ANSI C que expandía los argumentos formales de macro que aparecían dentro de literales de cadena y constantes de caracteres. El código que se basaba en esta extensión se debe reescribir mediante el operador de cadena # ( ).

Se omite el espacio en blanco que precede al primer token y sigue al último token del argumento real. Cualquier espacio en blanco entre los tokens del argumento real se reduce a un único espacio en blanco en el literal de cadena resultante. Por lo tanto, si se produce un comentario entre dos tokens en el argumento real, se reduce a un único espacio en blanco. El literal de cadena resultante se concatena automáticamente con los literales de cadena adyacentes separados solo por espacios en blanco.

Además, si un carácter contenido en el argumento normalmente requiere una secuencia de escape cuando se usa en un literal de cadena, por ejemplo, el carácter de comilla ( ) o barra diagonal inversa ( ), la barra diagonal inversa de escape necesaria se inserta automáticamente antes del `"` `\` carácter.

El operador de cadenas de Microsoft C++ no se comporta correctamente cuando se usa con cadenas que incluyen secuencias de escape. En esta situación, el compilador genera el error del compilador C2017.

Ejemplos

En el ejemplo siguiente se muestra una definición de macro que incluye el operador stringizing y una función main que invoca la macro:

```
// stringizer.cpp
#include <stdio.h>
#define stringer( x ) printf_s( #x "\n" )
int main() {
   stringer( In quotes in the printf function call );
   stringer( "In quotes when printed to the screen" );
   stringer( "This: \"  prints an escaped double quote" );
}
```

Las `stringer` macros se expanden durante el preprocesamiento, generando el código siguiente:

```
int main() {
   printf_s( "In quotes in the printf function call" "\n" );
   printf_s( "\"In quotes when printed to the screen\"" "\n" );
   printf_s( "\"This: \\\" prints an escaped double quote\"" "\n" );
}
```

```
In quotes in the printf function call
"In quotes when printed to the screen"
"This: \"  prints an escaped double quote"
```

En el ejemplo siguiente se muestra cómo se puede expandir un parámetro de macro:

```
// stringizer_2.cpp
// compile with: /E
#define F abc
#define B def
#define FB(arg) #arg
#define FB1(arg) FB(arg)
FB(F B)
FB1(F B)
```

Vea también

Operadores de preprocesador

Operador de generación de caracteres (#@)

13/08/2021 • 2 minutes to read

Específicos de Microsoft

El operador charizing solo puede utilizarse con argumentos de macros. Si precede a un parámetro formal en la definición de la macro, el argumento real se incluye entre comillas simples y se trata como un carácter cuando se `#@` expande la macro. Por ejemplo:

```
#define makechar(x)  #@x
```

hace que la instrucción

```
a = makechar(b);
```

se expanda a

```
a = 'b';
```

El carácter de comilla simple (`'`) no se puede usar con el operador de caracteres.

FIN de Específicos de Microsoft

Vea también

Operadores de preprocesador

Operador de pegado de token (##)

16/08/2021 • 2 minutes to read

El operador double-number-sign o *token-pasting* ( ), que a veces se denomina operador de combinación o combinación, se usa en macros de tipo objeto y de **##** tipo función. Permite que los tokens independientes se unan a un solo token y, por lo tanto, no puede ser el primer ni el último token de la definición de macro.

Si un parámetro formal en una definición de macro va precedido o seguido del operador de pegado de token, el parámetro formal se sustituye inmediatamente por el argumento real sin expandir. La expansión de la macro no se realiza en el argumento antes de la sustitución.

A continuación, se quita cada aparición del operador de pegar tokens en *la cadena* de token y se concatenan los tokens anteriores y siguientes. El token resultante debe ser un token válido. Si es válido, el token se analiza para una posible sustitución en caso de que represente un nombre de macro. El identificador representa el nombre por el que se conocerán los tokens concatenados en el programa antes de la sustitución. Cada token representa un token definido en alguna parte, ya sea dentro del programa o en la línea de comandos del compilador. El espacio en blanco que precede o que sigue al operador es opcional.

En este ejemplo se muestra el uso de los operadores de generación de cadenas y de pegado de token al especificar la salida del programa:

```
#define paster( n ) printf_s( "token" #n " = %d", token##n )
int token9 = 9;
```

Si se llama a una macro con un argumento numérico como

```
paster( 9 );
```

la macro produce

```
printf_s( "token" "9" " = %d", token9 );
```

que se convierte en

```
printf_s( "token9 = %d", token9 );
```

Ejemplo

```
// preprocessor_token_pasting.cpp
#include <stdio.h>
#define paster( n ) printf_s( "token" #n " = %d", token##n )
int token9 = 9;

int main()
{
   paster(9);
}
```

```
token9 = 9
```

Vea también

Operadores de preprocesador

Macros (C/C++)

16/08/2021 • 2 minutes to read

El preprocesador expande las macros en todas las líneas excepto las directivas de *preprocesador*, líneas que tienen como primer carácter que no es # de espacio en blanco. Expande las macros en partes de algunas directivas que no se omiten como parte de una compilación condicional. *Las directivas de compilación* condicional permiten suprimir la compilación de partes de un archivo de origen. Prueban una expresión constante o un identificador para determinar qué bloques de texto se van a pasar al compilador y cuáles quitar del archivo de origen durante el preprocesamiento.

La directiva `#define` se utiliza normalmente para asociar identificadores significativos con constantes, palabras clave e instrucciones o expresiones usadas con frecuencia. Los identificadores que representan constantes a veces se denominan constantes *simbólicas* o *constantes de manifiesto*. Los identificadores que representan instrucciones o expresiones se *denominan macros*. En esta documentación de preprocesador, solo se utiliza el término "macro".

Cuando se reconoce el nombre de una macro en el texto de origen del programa o en los argumentos de otros comandos de preprocesador, se trata como una llamada a esa macro. El nombre de la macro se reemplaza por una copia del cuerpo de la macro. Si la macro acepta argumentos, los argumentos reales que siguen al nombre de la macro se sustituyen por los parámetros formales en el cuerpo de la macro. El proceso de reemplazar una llamada macro por la copia procesada del cuerpo se denomina *expansión* de la llamada macro.

En la práctica, hay dos tipos de macros. *Las macros de tipo* objeto no toman ningún argumento. *Las macros de tipo* función se pueden definir para aceptar argumentos, de modo que parezcan llamadas de función y actúen como ellas. Dado que las macros no generan llamadas de función reales, a veces puede hacer que los programas se ejecuten más rápido reemplazando las llamadas de función por macros. (En C++, las funciones insertadas suelen ser un método preferido). Sin embargo, las macros pueden crear problemas si no los define y los usa con cuidado. Puede que tenga que utilizar paréntesis en definiciones de macro con argumentos para conservar la prioridad correcta en una expresión. Además, puede que las macros no controlen correctamente las expresiones con efectos secundarios. Para obtener más información, vea el `getrandom` ejemplo de la directiva #define .

Una vez que haya definido una macro, no puede volver a definirla a un valor diferente sin quitar primero la definición original. Sin embargo, puede volver a definir la macro con exactamente la misma definición. Por lo tanto, la misma definición puede aparecer más de una vez en un programa.

La `#undef` directiva quita la definición de una macro. Una vez que haya quitado la definición, puede volver a definir la macro a un valor diferente. La #define directiva y la directiva #undef de trabajo analizan las directivas y `#define` , `#undef` respectivamente.

Para obtener más información, vea

- Macros y C++
- Macros variádicas
- Macros predefinidas

Vea también

Referencia del preprocesador de C/C++

Macros y C++

16/08/2021 • 2 minutes to read

C++ ofrece nuevas funcionalidades, algunas de las cuales suplanta las que ofrece el preprocesador ANSI C. Estas nuevas capacidades mejoran la seguridad de tipos y la previsibilidad del lenguaje:

- En C++, los objetos declarados como `const` se pueden usar en expresiones constantes. Permite a los programas declarar constantes que tienen información de tipo y valor. Pueden declarar enumeraciones que se pueden ver de forma simbólica con el depurador. Cuando se usa la directiva de preprocesador para definir constantes, no es tan `#define` precisa y no es segura para tipos. No se asigna almacenamiento para un objeto, a `const` menos que el programa contenga una expresión que tome su dirección.
- La capacidad de función insertada de C++ suplanta las macros de tipo de función. Las ventajas de utilizar funciones insertadas respecto a macros son:
  - Seguridad de tipos. Las funciones insertadas están sujetas a la misma comprobación de tipos que las funciones normales. Las macros no son seguras para tipos.
  - Se corrige el control de argumentos que tienen efectos secundarios. Las funciones insertadas evalúan las expresiones proporcionadas como argumentos antes de especificar el cuerpo de la función. Por lo tanto, no hay ninguna posibilidad de que una expresión con efectos secundarios no sea segura.

Para obtener más información sobre las funciones insertadas, vea inline, __inline, _ _forceinline.

Por compatibilidad con versiones anteriores, todos los servicios de preprocesador que existían en ANSI C y en las especificaciones anteriores de C++ se conservan para Microsoft C++.

Vea también

Macros predefinidas
Macros (C/C++)

Macros variádicas

12/08/2021 • 2 minutes to read

Las macros variádicas son macros estilo funciones que contienen un número variable de argumentos.

Comentarios

Para usar macros variádicas, los puntos suspensivos se pueden especificar como argumento formal final en una definición de macro y el identificador de reemplazo se puede usar en la definición para insertar los `__VA_ARGS__` argumentos adicionales. `__VA_ARGS__` se reemplaza por todos los argumentos que coinciden con los puntos suspensivos, incluidas las comas entre ellos.

El estándar de C especifica que se debe pasar al menos un argumento a los puntos suspensivos para asegurarse de que la macro no se resuelve en una expresión con una coma final. La implementación tradicional de Microsoft C++ suprime una coma final si no se pasa ningún argumento a los puntos suspensivos. Cuando se establece la opción del compilador, no se suprime la `/Zc:preprocessor` coma final.

Ejemplo

```
// variadic_macros.cpp
#include <stdio.h>
#define EMPTY

#define CHECK1(x, ...) if (!(x)) { printf(__VA_ARGS__); }
#define CHECK2(x, ...) if ((x)) { printf(__VA_ARGS__); }
#define CHECK3(...) { printf(__VA_ARGS__); }
#define MACRO(s, ...) printf(s, __VA_ARGS__)

int main() {
    CHECK1(0, "here %s %s %s", "are", "some", "varargs1(1)\n");
    CHECK1(1, "here %s %s %s", "are", "some", "varargs1(2)\n");   // won't print

    CHECK2(0, "here %s %s %s", "are", "some", "varargs2(3)\n");   // won't print
    CHECK2(1, "here %s %s %s", "are", "some", "varargs2(4)\n");

    // always invokes printf in the macro
    CHECK3("here %s %s %s", "are", "some", "varargs3(5)\n");

    MACRO("hello, world\n");

    MACRO("error\n", EMPTY); // would cause error C2059, except VC++
                             // suppresses the trailing comma
}
```

```
here are some varargs1(1)
here are some varargs2(4)
here are some varargs3(5)
hello, world
error
```

Vea también

Macros (C/C++)

Macros predefinidas

23/08/2021 • 17 minutes to read

El compilador de C/C++ de Microsoft (MSVC) predefine ciertas macros de preprocesador en función del lenguaje (C o C++), el destino de compilación y las opciones del compilador elegidas.

MSVC admite las macros de preprocesador predefinidas que exigen los estándares C99, C11 y C17 de ANSI/ISO y los estándares C++14 y C++17 de ISO. La implementación también admite otras macros de preprocesador específicas de Microsoft. Algunas macros están definidas solo para entornos de compilación específicos o para opciones del compilador. Excepto donde se indique lo contrario, las macros se definen en una unidad de traducción como si se hubieran especificado como argumentos de la opción de compilador `/D`. Cuando están definidas, el preprocesador expande las macros hasta los valores especificados antes de la compilación. Estas macros predefinidas no toman ningún argumento y no se pueden volver a definir.

Identificador predefinido estándar

El compilador admite este identificador predefinido especificado por ISO C99 e ISO C++11.

- `__func__` Nombre no completo y sin adornar de la función de inclusión como una matriz **static const** de función local de `char`.

```
void example(){
    printf("%s\n", __func__);
} // prints "example"
```

Macros predefinidas estándar

El compilador admite estas macros predefinidas especificadas por los estándares ISO C99, C11, C17 e ISO C++17.

- `__cplusplus` Se define como un valor literal entero cuando la unidad de traducción se compila como C++. De lo contrario, no se define.
- `__DATE__` La fecha de compilación del archivo de código fuente actual. La fecha es un literal de cadena de longitud constante con el formato *Mmm dd aaaa*. El nombre del mes *Mmm* es el mismo que el nombre de mes abreviado que genera la función asctime de la biblioteca en tiempo de ejecución de C (CRT). El primer carácter de la fecha *dd* es un espacio si el valor es menor que 10. Esta macro siempre está definida.
- `__FILE__` Nombre del archivo de código fuente actual. `__FILE__` se expande a un literal de cadena de caracteres. Para garantizar que se muestra la ruta de acceso completa al archivo, use `/FC` (Ruta de acceso completa de archivo de código fuente en diagnósticos). Esta macro siempre está definida.
- `__LINE__` Se define como el número de línea entero del archivo de código fuente actual. El valor de la macro `__LINE__` se puede cambiar con una directiva `#line`. Esta macro siempre está definida.
- `__STDC__` Se define como 1 solo cuando se compila como C y si se especifica la opción del compilador `/Za`. De lo contrario, no se define.
- `__STDC_HOSTED__` Se define como 1 si la implementación es una *implementación hospedada* que admite la biblioteca estándar completa requerida. De lo contrario, se define como 0.

- `__STDC_NO_ATOMICS__` se define como 1 si la implementación no admite tipos atómicos estándar opcionales. La implementación de MSVC la define como 1 cuando se compila como C y se especifica una de las opciones C11 o C17 de `/std` .
- `__STDC_NO_COMPLEX__` se define como 1 si la implementación no admite números complejos estándar opcionales. La implementación de MSVC la define como 1 cuando se compila como C y se especifica una de las opciones C11 o C17 de `/std` .
- `__STDC_NO_THREADS__` se define como 1 si la implementación no admite subprocesos estándar opcionales. La implementación de MSVC la define como 1 cuando se compila como C y se especifica una de las opciones C11 o C17 de `/std` .
- `__STDC_NO_VLA__` se define como 1 si la implementación no admite matrices estándar de longitud variable. La implementación de MSVC la define como 1 cuando se compila como C y se especifica una de las opciones C11 o C17 de `/std` .
- `__STDC_VERSION__` se define cuando se compila como C y se especifica una de las opciones C11 o C17 de `/std` . Se expande a `201112L` para `/std:c11` , y a `201710L` para `/std:c17` .
- `__STDCPP_THREADS__` Se define como 1 solo si un programa puede tener más de un subproceso de ejecución y se compila como C++. De lo contrario, no se define.
- `__TIME__` Hora de traducción de la unidad de traducción preprocesada. La hora es un literal de cadena de caracteres con la forma *hh:mm:ss*, igual que la hora devuelta por la función asctime de CRT. Esta macro siempre está definida.

Macros predefinidas específicas de Microsoft

MSVC admite estas macros predefinidas adicionales.

- `__ATOM__` Se define como 1 cuando se establece la opción del compilador `/favor:ATOM` y el destino del compilador es x86 o x64. De lo contrario, no se define.
- `__AVX__` Se define como 1 cuando se establecen las opciones del compilador `/arch:AVX` `/arch:AVX2` , `/arch:AVX512` y el destino del compilador es x86 o x64. De lo contrario, no se define.
- `__AVX2__` Se define como 1 cuando se establecen las opciones del compilador `/arch:AVX2` o `/arch:AVX512` , y el destino del compilador es x86 o x64. De lo contrario, no se define.
- `__AVX512BW__` Se define como 1 cuando se establece la opción del compilador `/arch:AVX512` y el destino del compilador es x86 o x64. De lo contrario, no se define.
- `__AVX512CD__` Se define como 1 cuando se establece la opción del compilador `/arch:AVX512` y el destino del compilador es x86 o x64. De lo contrario, no se define.
- `__AVX512DQ__` Se define como 1 cuando se establece la opción del compilador `/arch:AVX512` y el destino del compilador es x86 o x64. De lo contrario, no se define.
- `__AVX512F__` Se define como 1 cuando se establece la opción del compilador `/arch:AVX512` y el destino del compilador es x86 o x64. De lo contrario, no se define.
- `__AVX512VL__` Se define como 1 cuando se establece la opción del compilador `/arch:AVX512` y el destino del compilador es x86 o x64. De lo contrario, no se define.
- `_CHAR_UNSIGNED` Se define como 1 si el tipo `char` predeterminado no tiene signo. Este valor se define cuando se establece la opción del compilador `/J` (El tipo de carácter predeterminado no tiene signo). De lo contrario, no se define.

- `__CLR_VER` Se define como un literal entero que representa la versión de Common Language Runtime (CLR) utilizada para compilar la aplicación. El valor se codifica con la forma `Mmmbbbbb`, donde `M` es la versión principal del runtime, `mm` es la versión secundaria del runtime y `bbbbb` es el número de compilación. `__CLR_VER` se define si se establece la opción del compilador `/clr` . De lo contrario, no se define.

  ```
  // clr_ver.cpp
  // compile with: /clr
  using namespace System;
  int main() {
     Console::WriteLine(__CLR_VER);
  }
  ```

- `_CONTROL_FLOW_GUARD` Se define como 1 cuando se establece la opción del compilador `/guard:cf` (Habilitar protección del flujo de control). De lo contrario, no se define.

- `__COUNTER__` Se expande a un literal entero que comienza en 0. El valor se incrementa en 1 cada vez que se usa en un archivo de código fuente o en encabezados incluidos del archivo de código fuente. `__COUNTER__` recuerda su estado cuando se usan encabezados precompilados. Esta macro siempre está definida.

  En este ejemplo se usa `__COUNTER__` para asignar identificadores únicos a tres objetos diferentes del mismo tipo. El constructor `exampleClass` toma un entero como parámetro. En `main`, la aplicación declara tres objetos de tipo `exampleClass`, usando `__COUNTER__` como parámetro de identificador único:

```
// macro__COUNTER__.cpp
// Demonstration of __COUNTER__, assigns unique identifiers to
// different objects of the same type.
// Compile by using: cl /EHsc /W4 macro__COUNTER__.cpp
#include <stdio.h>

class exampleClass {
    int m_nID;
public:
    // initialize object with a read-only unique ID
    exampleClass(int nID) : m_nID(nID) {}
    int GetID(void) { return m_nID; }
};

int main()
{
    // __COUNTER__ is initially defined as 0
    exampleClass e1(__COUNTER__);

    // On the second reference, __COUNTER__ is now defined as 1
    exampleClass e2(__COUNTER__);

    // __COUNTER__ is now defined as 2
    exampleClass e3(__COUNTER__);

    printf("e1 ID: %i\n", e1.GetID());
    printf("e2 ID: %i\n", e2.GetID());
    printf("e3 ID: %i\n", e3.GetID());

    // Output
    // ------------------------------
    // e1 ID: 0
    // e2 ID: 1
    // e3 ID: 2

    return 0;
}
```

- `__cplusplus_cli` Se define como el valor literal entero 200406 cuando se compila como C++ y se establece una opción del compilador `/clr` . De lo contrario, no se define. Cuando se define, `__cplusplus_cli` está activo en la unidad de traducción.

```
// cplusplus_cli.cpp
// compile by using /clr
#include "stdio.h"
int main() {
   #ifdef __cplusplus_cli
      printf("%d\n", __cplusplus_cli);
   #else
      printf("not defined\n");
   #endif
}
```

- `__cplusplus_winrt` Se define como el valor literal entero 201009 cuando se compila como C++ y se establece la opción del compilador `/ZW` (Compilación de Windows Runtime). De lo contrario, no se define.
- `_CPPRTTI` Se define como 1 si se establece la opción del compilador `/GR` (Habilitar la información de tipo en tiempo de ejecución). De lo contrario, no se define.
- `_CPPUNWIND` Se define como 1 si se establecen una o más de las opciones del compilador `/GX` (Habilitar el control de excepciones), `/clr` (Compilación de Common Language Runtime) o `/EH` (Modelo de

control de excepciones). De lo contrario, no se define.

- `_DEBUG` Se define como 1 cuando se establecen las opciones del compilador `/LDd` `/MDd`, o `/MTd`. De lo contrario, no se define.
- `_DLL` Se define como 1 cuando se establecen las opciones del compilador `/MD` o `/MDd` (DLL multiproceso). De lo contrario, no se define.
- `__FUNCDNAME__` Se define como literal de cadena que contiene el nombre representativo de la función de inclusión. La macro solo se define dentro de una función. La macro `__FUNCDNAME__` no se expande si se usan las opciones del compilador `/EP` o `/P` .

  En este ejemplo se usan las macros `__FUNCDNAME__` , `__FUNCSIG__` y `__FUNCTION__` para mostrar la información de la función.

  ```
  // Demonstrates functionality of __FUNCTION__, __FUNCDNAME__, and __FUNCSIG__ macros
  void exampleFunction()
  {
      printf("Function name: %s\n", __FUNCTION__);
      printf("Decorated function name: %s\n", __FUNCDNAME__);
      printf("Function signature: %s\n", __FUNCSIG__);

      // Sample Output
      // -----------------------------------------------
      // Function name: exampleFunction
      // Decorated function name: ?exampleFunction@@YAXXZ
      // Function signature: void __cdecl exampleFunction(void)
  }
  ```

- `__FUNCSIG__` Se define como literal de cadena que contiene la firma de la función de inclusión. La macro solo se define dentro de una función. La macro `__FUNCSIG__` no se expande si se usan las opciones del compilador `/EP` o `/P` . Cuando se compila para un destino de 64 bits, la convención de llamada es `__cdecl` de forma predeterminada. Para obtener un ejemplo de uso, vea la macro `__FUNCDNAME__` .
- `__FUNCTION__` Se define como literal de cadena que contiene el nombre no representativo de la función de inclusión. La macro solo se define dentro de una función. La macro `__FUNCTION__` no se expande si se usan las opciones del compilador `/EP` o `/P` . Para obtener un ejemplo de uso, vea la macro `__FUNCDNAME__` .
- `_INTEGRAL_MAX_BITS` Se define como el valor literal entero 64, el tamaño máximo (en bits) para un tipo entero no vector. Esta macro siempre está definida.

  ```
  // integral_max_bits.cpp
  #include <stdio.h>
  int main() {
      printf("%d\n", _INTEGRAL_MAX_BITS);
  }
  ```

- `__INTELLISENSE__` Se define como 1 durante un paso del compilador de IntelliSense en el IDE de Visual Studio. De lo contrario, no se define. Puede usar esta macro para proteger el código que el compilador de IntelliSense no entiende o para alternar entre la compilación y el compilador de IntelliSense. Para más información, vea Sugerencias para la solución de problemas de ralentización de IntelliSense.
- `_ISO_VOLATILE` Se define como 1 si se establece la opción del compilador `/volatile:iso` . De lo contrario, no se define.
- `_KERNEL_MODE` Se define como 1 si se establece la opción del compilador `/kernel` (Crear binario en modo

kernel). De lo contrario, no se define.

- `_M_AMD64` Se define como el valor literal entero 100 para las compilaciones destinadas a procesadores x64. De lo contrario, no se define.
- `_M_ARM` Se define como el valor literal entero 7 para las compilaciones destinadas a procesadores ARM. De lo contrario, no se define.
- `_M_ARM_ARMV7VE` Se define como 1 cuando se establece la opción del compilador `/arch:ARMv7VE` para las compilaciones destinadas a procesadores ARM. De lo contrario, no se define.
- `_M_ARM_FP` Se define como un valor literal entero que indica qué opción del compilador `/arch` se estableció para destinos de procesador ARM. De lo contrario, no se define.
  - Valor en el rango entre 30 y 39 si no se especificó ninguna opción `/arch` de ARM, lo que indica que se usó la arquitectura predeterminada para la ARM (`VFPv3`).
  - Valor en el rango entre 40 y 49 si se estableció `/arch:VFPv4` .
  - Para obtener más información, consulte `/arch` (ARM).
- `_M_ARM64` Se define como 1 para las compilaciones destinadas a procesadores ARM de 64 bits. De lo contrario, no se define.
- `_M_CEE` Se define como 001 si se establece alguna opción del compilador `/clr` (Compilación de Common Language Runtime). De lo contrario, no se define.
- `_M_CEE_PURE` En desuso a partir de Visual Studio 2015. Se define como 001 si se establece la opción del compilador `/clr:pure` . De lo contrario, no se define.
- `_M_CEE_SAFE` En desuso a partir de Visual Studio 2015. Se define como 001 si se establece la opción del compilador `/clr:safe` . De lo contrario, no se define.
- `_M_FP_EXCEPT` Se define como 1 si se establecen las opciones del compilador `/fp:except` o `/fp:strict` . De lo contrario, no se define.
- `_M_FP_FAST` Se define como 1 si se establece la opción del compilador `/fp:fast` . De lo contrario, no se define.
- `_M_FP_PRECISE` Se define como 1 si se establece la opción del compilador `/fp:precise` . De lo contrario, no se define.
- `_M_FP_STRICT` Se define como 1 si se establece la opción del compilador `/fp:strict` . De lo contrario, no se define.
- `_M_IX86` Se define como el valor literal entero 600 para las compilaciones destinadas a procesadores x86. Esta macro no está definida para los destinos de compilación x64 o ARM.
- `_M_IX86_FP` Se define como un valor literal entero que indica la opción del compilador `/arch` que se estableció, o el valor predeterminado. Esta macro siempre se define cuando el destino de compilación es un procesador x86. De lo contrario, no se define. Cuando se define, el valor es:
  - 0 si la opción del compilador `/arch:IA32` se ha establecido.
  - 1 si la opción del compilador `/arch:SSE` se ha establecido.
  - 2 si la opción del compilador `/arch:SSE2` , `/arch:AVX` , `/arch:AVX2` o `/arch:AVX512` se ha establecido. Este valor es el predeterminado si no se ha especificado una opción del compilador `/arch` . Cuando se especifica `/arch:AVX` , la macro `__AVX__` también se define. Cuando se especifica `/arch:AVX2` , también se definen `__AVX__` y `__AVX2__` . Cuando se especifica

`/arch:AVX512`, también se definen `__AVX__`, `__AVX2__`, `__AVX512BW__`, `__AVX512CD__`, `__AVX512DQ__`, `__AVX512F__` y `__AVX512VL__`.

  - Para obtener más información, vea `/arch` (x86).

- `_M_X64` Se define como el valor literal entero 100 para las compilaciones destinadas a procesadores x64. De lo contrario, no se define.

- `_MANAGED` Se define como 1 cuando se establece la opción del compilador `/clr`. De lo contrario, no se define.

- `_MSC_BUILD` Se define como un literal entero que contiene el elemento de número de revisión del número de versión del compilador. El número de revisión es el cuarto elemento del número de versión delimitado por puntos. Por ejemplo, si el número de versión del compilador de Microsoft C/C++ es 15.00.20706.01, la macro `_MSC_BUILD` se evalúa como 1. Esta macro siempre está definida.

- `_MSC_EXTENSIONS` Se define como 1 si se establece la opción del compilador `/Ze` (Habilitar extensiones de lenguaje). De lo contrario, no se define.

- `_MSC_FULL_VER` Se define como un literal de entero que codifica los elementos principal, secundario y de compilación del número de versión del compilador. El número principal es el primer elemento del número de versión delimitado por puntos, el número secundario es el segundo elemento y el número de compilación es el tercer elemento. Por ejemplo, si el número de versión del compilador de Microsoft C/C++ es 15.00.20706.01, la macro `_MSC_FULL_VER` se evalúa como 150020706. Escriba `cl /?` en la línea de comandos para ver el número de versión del compilador. Esta macro siempre está definida.

- `_MSC_VER` Se define como un literal entero que codifica los elementos principal y secundario del número de versión del compilador. El número principal es el primer elemento del número de versión delimitado por puntos y el número secundario es el segundo elemento. Por ejemplo, si el número de versión del compilador de Microsoft C/C++ es 17.00.51106.1, la macro `_MSC_VER` se evalúa como 1700. Escriba `cl /?` en la línea de comandos para ver el número de versión del compilador. Esta macro siempre está definida.

VERSIÓN DE VISUAL STUDIO	`_MSC_VER`
Visual Studio 6.0	1200
Visual Studio .NET 2002 (7.0)	1300
Visual Studio .NET 2003 (7.1)	1310
Visual Studio 2005 (8.0)	1400
Visual Studio 2008 (9.0)	1.500
Visual Studio 2010 (10.0)	1600
Visual Studio 2012 (11.0)	1700
Visual Studio 2013 (12.0)	1800
Visual Studio 2015 (14.0)	1900
Visual Studio 2017 RTW (15.0)	1910

VERSIÓN DE VISUAL STUDIO	`_MSC_VER`
Visual Studio 2017 versión 15.3	1911
Versión 15.5 de Visual Studio 2017	1912
Visual Studio 2017, versión 15.6	1913
Visual Studio 2017 versión 15.7	1914
Visual Studio 2017, versión 15.8	1915
Visual Studio 2017, versión 15.9	1916
Visual Studio 2019 RTW (16.0)	1920
Visual Studio 2019, versión 16.1	1921
Visual Studio 2019, versión 16.2	1922
Visual Studio 2019 versión 16.3	1923
Visual Studio 2019 versión 16.4	1924
Visual Studio 2019, versión 16.5	1925
Visual Studio 2019 versión 16.6	1926
Visual Studio 2019, versión 16.7	1927
Visual Studio 2019, versión 16.8, 16.9	1928
Visual Studio 2019, versión 16.10, 16.11	1929

Para probar las versiones o las actualizaciones del compilador en una versión concreta de Visual Studio o posterior, use el operador `>=`. Puede usarlo en una directiva condicional para comparar `_MSC_VER` con esa versión conocida. Si quiere comparar varias versiones mutuamente excluyentes, ordene las comparaciones por orden descendiente de número de versión. Por ejemplo, este código comprueba los compiladores publicados en Visual Studio 2017 y versiones posteriores. A continuación, comprueba los compiladores publicados en Visual Studio 2015 o versiones posteriores. Después, comprueba todos los compiladores publicados antes de Visual Studio 2015:

```
#if _MSC_VER >= 1910
// . . .
#elif _MSC_VER >= 1900
// . . .
#else
// . . .
#endif
```

Para comprobar las versiones del compilador que comparten números principales y secundarios, use los números principal, secundario y de compilación en `_MSC_FULL_VER` para las comparaciones. Los compiladores de Visual Studio 2019, versión 16.9, tienen un valor `_MSC_FULL_VER` de 192829500 o

superior. Los compiladores de Visual Studio 2019, versión 16.11, tienen un valor `_MSC_FULL_VER` de 192930100 o superior.

Para obtener más información, vea Versión del compilador de Visual C++ en el blog del equipo de Microsoft C++.

- `_MSVC_LANG` Se define como un literal entero que especifica el estándar de lenguaje de C++ de destino del compilador. Se establece solo en código compilado como C++. La macro es el valor literal entero `201402L` de forma predeterminada o cuando se especifica la opción del compilador `/std:c++14`. La macro se establece en `201703L` si se especifica la opción del compilador `/std:c++17`. La macro se establece en `202002L` si se especifica la opción del compilador `/std:c++20`. Se establece en un valor más alto y sin especificar cuando se especifica la opción `/std:c++latest`. De lo contrario, la macro no se define. La macro `_MSVC_LANG` y las opciones del compilador `/std` (Especificar la versión estándar del lenguaje) están disponibles a partir de Visual Studio 2015 Update 3.
- `__MSVC_RUNTIME_CHECKS` Se define como 1 cuando se establece una de las opciones del compilador `/RTC`. De lo contrario, no se define.
- `_MSVC_TRADITIONAL`:
  - Disponible a partir de la versión 15.8 de Visual Studio 2017: Se define como 0 cuando se establece la opción del compilador `/experimental:preprocessor` del modo de conformidad del preprocesador. Se define como 1 de forma predeterminada, o cuando se establece la opción del compilador `/experimental:preprocessor-`, para indicar que está en uso el preprocesador tradicional.
  - Disponible a partir de la versión 16.5 de Visual Studio 2019: Se define como 0 cuando se establece la opción del compilador `/Zc:preprocessor` del modo de conformidad del preprocesador. Se define como 1 de forma predeterminada, o cuando se establece la opción del compilador `/Zc:preprocessor-`, para indicar que el preprocesador tradicional está en uso (básicamente, `/Zc:preprocessor` reemplaza a `/experimental:preprocessor`, que está en desuso).

```
#if defined(_MSVC_TRADITIONAL) && _MSVC_TRADITIONAL
// Logic using the traditional preprocessor
#else
// Logic using cross-platform compatible preprocessor
#endif
```

- `_MT` Se define como 1 cuando se especifica `/MD` o `/MDd` (DLL multiproceso), o bien `/MT` o `/MTd` (Multiproceso). De lo contrario, no se define.
- `_NATIVE_WCHAR_T_DEFINED` Se define como 1 cuando se establece la opción del compilador `/Zc:wchar_t`. De lo contrario, no se define.
- `_OPENMP` Se define como literal entero 200203, si se establece la opción del compilador `/openmp` (Habilitar la compatibilidad con OpenMP 2.0). Este valor representa la fecha de la especificación de OpenMP implementada por MSVC. De lo contrario, no se define.

```
// _OPENMP_dir.cpp
// compile with: /openmp
#include <stdio.h>
int main() {
    printf("%d\n", _OPENMP);
}
```

- `_PREFAST_` Se define como 1 cuando se establece la opción del compilador `/analyze`. De lo contrario, no se define.

- `__SANITIZE_ADDRESS__` Disponible a partir de la versión 16.9 de Visual Studio 2019. Se define como 1 cuando se establece la opción del compilador `/fsanitize=address` . De lo contrario, no se define.
- `__TIMESTAMP__` Se define como un literal de cadena que contiene la fecha y hora de la última modificación del archivo de código fuente actual, en el formato de longitud constante abreviado devuelto por la función `asctime` de CRT; por ejemplo, `Fri 19 Aug 13:32:58 2016`. Esta macro siempre está definida.
- `_VC_NODEFAULTLIB` Se define como 1 cuando se establece la opción del compilador `/Zl` (Omitir nombre de biblioteca predeterminada). De lo contrario, no se define.
- `_WCHAR_T_DEFINED` Se define como 1 cuando se establece la opción del compilador `/Zc:wchar_t` predeterminada. La macro `_WCHAR_T_DEFINED` está definida, pero no tiene ningún valor si se establece la opción del compilador `/Zc:wchar_t-` y `wchar_t` se define en un archivo de encabezado del sistema incluido en el proyecto. De lo contrario, no se define.
- `_WIN32` Se define como 1 cuando el destino de compilación es ARM de 32 bits, ARM de 64 bits, x86 o x64. De lo contrario, no se define.
- `_WIN64` Se define como 1 cuando el destino de compilación es ARM de 64 bits o x64. De lo contrario, no se define.
- `_WINRT_DLL` Se define como 1 cuando se compila como C++ y se establecen las opciones del compilador `/ZW` (Compilación de Windows Runtime) y `/LD` , o `/LDd` . De lo contrario, no se define.

No hay ninguna macro de preprocesador que identifique la versión de la biblioteca ATL o MFC predefinida por el compilador. Los encabezados de la biblioteca ATL y MFC definen internamente estas macros de versión. No están definidas en las directivas de preprocesador creadas antes de incluir el encabezado necesario.

- `_ATL_VER` Se define en <atldef.h> como un literal entero que codifica el número de versión de ATL.
- `_MFC_VER` Se define en <afxver_.h> como un literal entero que codifica el número de versión de MFC.

Vea también

Macros (C/C++)
Operadores de preprocesador
Directivas de preprocesador

Resumen de la gramática del preprocesador (C/C++)

13/08/2021 • 2 minutes to read

En este artículo se describe la gramática formal del preprocesador de C y C++. Trata la sintaxis de los operadores y directivas de preprocesamiento. Para obtener más información, vea Directivas de preprocesador y Pragma y las `__pragma` palabras clave `_Pragma` y.

Definiciones del resumen de gramática

Los elementos terminales son puntos de conexión en una definición de sintaxis. No hay ninguna otra posible resolución. Los elementos terminales incluyen el conjunto de palabras reservadas e identificadores definidos por el usuario.

Los elementos no terminales son marcadores en la sintaxis. La mayoría se definen en otro lugar en este resumen de la sintaxis. Las definiciones pueden ser recursivas. Los siguientes elementos no terminales se definen en la sección Convenciones léxicas de la Referencia *del lenguaje C++:*

`constant` , `constant-expression` , `identifier` , `keyword` , `operator` , `punctuator`

Un componente opcional se indica mediante el subíndice $_{opt}$. Por ejemplo, la sintaxis siguiente indica una expresión opcional entre llaves:

{ *expression* $_{opt}$ }

Convenciones de documento

Las convenciones utilizan distintos atributos de fuente de diferentes componentes de la sintaxis. Los símbolos y fuentes son los siguientes:

ATRIBUTO	DESCRIPCIÓN
nonterminal	La cursiva indica elementos no terminales.
`#include`	Los elementos terminales en negrita son símbolos y palabras literales reservadas que deben especificarse tal como aparecen. Los caracteres en este contexto siempre distinguen entre mayúsculas y minúsculas.
opt	Los elementos no terminales seguidos de $_{opt}$ son siempre opcionales.
tipo de letra predeterminado	Los caracteres del juego descrito o mostrado en este tipo de letra se pueden usar como elementos terminales en las instrucciones.

Un signo de dos puntos (`:`) que sigue a un elemento no terminal presenta su definición. Las definiciones alternativas se indican en líneas distintas.

En los bloques de sintaxis de código, estos símbolos en el tipo de letra predeterminado tienen un significado especial:

SÍMBOLO	DESCRIPCIÓN
[ ]	Los corchetes rodean un elemento opcional.
{ | }	Las llaves rodean elementos alternativos, separados por barras verticales.
...	Indica que se puede repetir el patrón de elemento anterior.

En los bloques de sintaxis de código, las comas ( ), los puntos ( ), los puntos y comas ( ), los dos puntos ( ), los paréntesis ( ), las comillas dobles ( ) y las comillas simples ( ) son `,` `.` `;` `:` `( )` `"` `'` literales.

Gramática del preprocesador

control-line:

- `#define` *identifier* *token-string* opt
- `#define` *identifier* `(` *identifier* opt `,` ... `,` *identifier* `)` opt *token-string* opt
- `#include` `"` *path-spec* `"`
- `#include` `<` *path-spec* `>`
- `#line` *digit-sequence* `"` *filename* `"` opt
- `#undef` *identifier*
- `#error` *token-string*
- `#pragma` *token-string*

constant-expression:

- `defined(` *identifier* `)`
- `defined` *identifier*
- cualquier otra expresión constante

conditional:

- *if-part* *elif-parts* opt *else-part* opt *endif-line*

if-part:

- *if-line* *text*

if-line:

- `#if` *constant-expression*
- `#ifdef` *identifier*
- `#ifndef` *identifier*

elif-parts:

- *elif-line* *text*
- *elif-parts* *elif-line* *text*

elif-line:

- `#elif` *constant-expression*

else-part:

- *else-line* *text*

else-line:

- `#else`

endif-line:

- `#endif`

`digit-sequence`:

`digit`

`digit-sequence` `digit`

`digit` : uno de

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

`token-string`:

Cadena de `token`

`token`:

`keyword`

`identifier`

`constant`

`operator`

`punctuator`

`filename`:

nombre de archivo del sistema operativo válido

`path-spec`:

Ruta de acceso válida

`text`:

cualquier secuencia de texto

> **NOTE**
>
> Los siguientes elementos no terminales se expanden en la sección Convenciones léxicas de la Referencia del lenguaje *C++:* `constant` `,,,,` y `constant-expression` `identifier` `keyword` `operator` `punctuator` .

Vea también

Referencia del preprocesador de C/C++

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

16/08/2021 • 4 minutes to read

Las directivas pragma especifican características del compilador específicas del equipo o específicas del sistema operativo. La palabra clave , que es específica del compilador de Microsoft, le permite codificar directivas dentro `__pragma` pragma de las definiciones de macro. El operador `_Pragma` de preprocesador estándar se introdujo en C99 y fue adoptado por C++11.

Sintaxis

** `#pragma` ** *token-string*
** `__pragma(` ** *token-string* `)` dos caracteres de subrayado iniciales: extensión específica de Microsoft
** `_Pragma(` ** *string-literal* `)` C99

Comentarios

Cada implementación de C y C++ admite algunas características exclusivas del equipo host o del sistema operativo. Algunos programas, por ejemplo, deben tener un control preciso sobre la ubicación de los datos en memoria o controlar la forma en que determinadas funciones reciben parámetros. Las directivas ofrecen una manera de que cada compilador ofrezca características específicas del sistema operativo y del equipo, al tiempo que mantienen la compatibilidad general con los `#pragma` lenguajes C y C++.

Las directivas pragma son específicas de la máquina o específicas del sistema operativo por definición y suelen ser diferentes para cada compilador. Se pragma puede usar en una directiva condicional para proporcionar una nueva funcionalidad de preprocesador. O bien, use uno para proporcionar información definida por la implementación al compilador.

La *cadena de token* es una serie de caracteres que representan una instrucción y argumentos del compilador específicos, si los hay. El signo de número ( ) debe ser el primer carácter que no sea de espacio en `#` blanco en la línea que contiene pragma . Los caracteres de espacio en blanco pueden separar el signo de número y la palabra " pragma ". A `#pragma` continuación, escriba cualquier texto que el traductor pueda analizar como tokens de preprocesamiento. El argumento de `#pragma` está sujeto a la expansión de macros.

El *literal de cadena* es la entrada de `_Pragma` . Se quitan las comillas externas y los espacios en blanco iniciales y finales. `\"` se reemplaza por `"` y se reemplaza por `\\` `\` .

El compilador emite una advertencia cuando encuentra un que pragma no reconoce y continúa la compilación.

Los compiladores de Microsoft C y C++ reconocen las pragma siguientes directivas:

`alloc_text`
`auto_inline`
`bss_seg`
`check_stack`
`code_seg`
`comment`
`component`
`conform` [1]

`const_seg`
`data_seg`
`deprecated`

`detect_mismatch`
`endregion`
`fenv_access`
`float_control`
`fp_contract`
`function`
`hdrstop`
`include_alias`
`init_seg` [1]
`inline_depth`
`inline_recursion`

`intrinsic`
`loop` [1]
`make_public`
`managed`
`message`
`omp`
`once`
`optimize`
`pack`
`pointers_to_members` [1]
`pop_macro`

`push_macro`
`region`
`runtime_checks`
`section`
`setlocale`
`strict_gs_check`
`system_header`
`unmanaged`
`vtordisp` [1]
`warning`

[1] Solo compatible con el compilador de C++.

Directivas pragma y opciones del compilador

Algunas pragma directivas proporcionan la misma funcionalidad que las opciones del compilador. Cuando se pragma alcanza un objeto en el código fuente, invalida el comportamiento especificado por la opción del compilador. Por ejemplo, si especificó , puede invalidar esta configuración del compilador `/Zp8` para secciones específicas del código con `pack` :

```
cl /Zp8 some_file.cpp
```

```
// some_file.cpp - packing is 8
// ...
#pragma pack(push, 1) - packing is now 1
// ...
#pragma pack(pop) - packing is 8 again
// ...
```

Palabra `__pragma` clave

El compilador también admite la palabra clave específica de `__pragma` Microsoft, que tiene la misma funcionalidad que la `#pragma` directiva . La diferencia es que la `__pragma` palabra clave se puede usar en línea en una definición de macro. La directiva no es utilizable en una definición de macro, porque el compilador interpreta el carácter de signo de número ('#') de la directiva como el operador de cadena `#pragma` (#).

En el ejemplo de código siguiente se muestra cómo se puede usar la palabra `__pragma` clave en una macro. Este código se ha extracto del encabezado *mfcdual.h* en el ejemplo ACDUAL de "Ejemplos de compatibilidad com del compilador":

```
#define CATCH_ALL_DUAL \
CATCH(COleException, e) \
{ \
_hr = e->m_sc; \
} \
AND_CATCH_ALL(e) \
{ \
__pragma(warning(push)) \
__pragma(warning(disable:6246)) /*disable _ctlState prefast warning*/ \
AFX_MANAGE_STATE(pThis->m_pModuleState); \
__pragma(warning(pop)) \
_hr = DualHandleException(_riidSource, e); \
} \
END_CATCH_ALL \
return _hr; \
```

Operador `_Pragma` de preprocesamiento

`_Pragma` es similar a la palabra clave específica de `__pragma` Microsoft. Se introdujo en el estándar de C en C99 y el estándar de C++ en C++11. Solo está disponible en C cuando se especifica la `/std:c11` opción o `/std:c17` . Para C++, está disponible en todos los `/std` modos, incluido el predeterminado.

A `__pragma` diferencia de , permite colocar directivas en una `_Pragma` pragma definición de macro. El literal de cadena debe ser el que se colocaría de otro modo después de una `#pragma` instrucción . Por ejemplo:

```
#pragma message("the #pragma way")
_Pragma ("message( \"the _Pragma way\")")
```

Las comillas y las barras diagonales hacia atrás deben ser de escape, como se muestra anteriormente. Se pragma omite una cadena que no se reconoce.

En el ejemplo de código siguiente se muestra cómo se podría usar la palabra `_Pragma` clave en una macro de tipo aserción. Crea una directiva pragma que suprime una advertencia cuando la expresión de condición es constante.

La definición de macro usa la expresión para macros de varias instrucciones para que se pueda usar como si `do` - `while(0)` fuera una instrucción. Para obtener más información, vea Macro de varias líneas de C en Stack

Overflow. La `_Pragma` instrucción solo se aplica a la línea de código que la sigue.

```
// Compile with /W4

#include <stdio.h>
#include <stdlib.h>

#define MY_ASSERT(BOOL_EXPRESSION) \
    do { \
        _Pragma("warning(suppress: 4127)") /* C4127 conditional expression is constant */  \
        if (!(BOOL_EXPRESSION)) {   \
            printf("MY_ASSERT FAILED: \"" #BOOL_EXPRESSION "\" on %s(%d)", __FILE__, __LINE__); \
            exit(-1); \
        } \
    } while (0)

int main()
{
    MY_ASSERT(0 && "Note that there is no warning: C4127 conditional expression is constant");

    return 0;
}
```

Vea también

Referencia del preprocesador de C/C++
Directivas de C pragma
Palabras clave

`alloc_text` pragma

14/08/2021 • 2 minutes to read

Denomina la sección de código donde se colocan las definiciones de función especificadas. debe pragma producirse entre un declarador de función y la definición de función para las funciones con nombre.

Sintaxis

```
#pragma alloc_text( "text-section" , function_1 [ , function_2 ... ] )
```

Comentarios

no `alloc_text` pragma controla las funciones miembro de C++ ni las funciones sobrecargadas. Solo es aplicable a las funciones declaradas con vinculación de C, es decir, a las funciones declaradas con la `extern "C"` especificación de vinculación. Si intenta usar esto en pragma una función con vinculación de C++, se genera un error del compilador.

Puesto que no se admite el direccionamiento de función mediante `__based` , la especificación de ubicaciones de sección requiere el uso de `alloc_text` pragma . El nombre especificado por *text-section* debe ir entre comillas dobles.

debe aparecer después de las declaraciones de cualquiera de las funciones especificadas y antes `alloc_text` pragma de las definiciones de estas funciones.

Las funciones a las que `alloc_text` pragma se hace referencia en se deben definir en el mismo módulo que pragma . De lo contrario, si una función no definida se compila más adelante en una sección de texto diferente, el error puede o no capturarse. Aunque el programa normalmente se ejecutará correctamente, la función no se asignará en las secciones previstas.

Otras limitaciones `alloc_text` de son las siguientes:

- No se puede usar dentro de una función.
- Debe utilizarse una vez declarada la función, pero antes de que esta se haya definido.

Vea también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

`auto_inline` pragma

17/08/2021 • 2 minutes to read

Excluye las funciones definidas dentro del intervalo donde se especifica para que no se consideren `off` candidatas para la expansión insertada automática.

Sintaxis

```
#pragma auto_inline( [ { on | off } ] )
```

Comentarios

Para usar `auto_inline` pragma , colómelo antes e inmediatamente después, no dentro, de una definición de función. su pragma efecto se produce tan pronto como se ve la primera definición pragma de función después de que se ve .

Vea también

Directivas Pragma y las palabras `__pragma` `_Pragma` clave y

`bss_seg` pragma

16/08/2021 • 2 minutes to read

Especifica la sección (segmento) en la que las variables no inicializadas se almacenan en el archivo de objeto (.obj).

Sintaxis

```
#pragma bss_seg( [ "section-name" [ , "section-class" ] ] )
#pragma bss_seg( { push | pop } [ , identifier ] [ , "section-name" [ , "section-class" ] ] )
```

Parámetros

`push`
(Opcional) Coloca un registro en la pila interna del compilador. puede `push` tener un identificador *y* un nombre *de sección.*

`pop`
(Opcional) Quita un registro de la parte superior de la pila interna del compilador. puede `pop` tener un identificador *y* un nombre *de sección.* Puede abrir varios registros con un solo `pop` comando mediante el *identificador*. *Section-name se* convierte en el nombre de la sección BSS activa después del pop.

identifier
(Opcional) Cuando se usa `push` con , asigna un nombre al registro en la pila interna del compilador. Cuando se usa con `pop` , la directiva quita los registros de la pila interna hasta que se *quita* el identificador. Si *no* se encuentra el identificador en la pila interna, no se hace nada.

"section-name"
(Opcional) Nombre de una sección. Cuando se usa `pop` con , la pila se abre y *section-name* se convierte en el nombre de sección de BSS activo.

"section-class"
(Opcional) Se omite, pero se incluye por compatibilidad con versiones de Microsoft C++ anteriores a la versión 2.0.

Comentarios

Una *sección* de un archivo de objeto es un bloque con nombre de datos que se carga en memoria como una unidad. Una *sección BSS* es una sección que contiene datos sin inicializar. En este artículo, los *términos segmento* y *sección* tienen el mismo significado.

La directiva indica al compilador que coloque todos los elementos de datos sin inicializar de la unidad de traducción en una sección `bss_seg` pragma de BSS denominada *section-name*. En algunos casos, el uso de puede acelerar los tiempos de carga mediante la agrupación de `bss_seg` datos no inicializados en una sección. De forma predeterminada, la sección BSS que se usa para los datos sin inicializar en un archivo de objeto se denomina `.bss` . Una `bss_seg` pragma directiva sin un parámetro *section-name* restablece el nombre de la sección BSS para los elementos de datos no inicializados posteriores en `.bss` .

Los datos asignados mediante `bss_seg` pragma no conservan ninguna información sobre su ubicación.

Para obtener una lista de nombres que no se deben usar para crear una sección, vea `/SECTION` .

También puede especificar secciones para los datos inicializados (`data_seg`), las funciones ( ) y las variables `code_seg` const (`const_seg`).

Puede usar la aplicación DUMPBIN.EXE para ver los archivos de objeto. Las versiones de DUMPBIN para cada arquitectura de destino admitida se incluyen con Visual Studio.

Ejemplo

```
// pragma_directive_bss_seg.cpp
int i;                      // stored in .bss
#pragma bss_seg(".my_data1")
int j;                      // stored in .my_data1

#pragma bss_seg(push, stack1, ".my_data2")
int l;                      // stored in .my_data2

#pragma bss_seg(pop, stack1)   // pop stack1 from stack
int m;                      // stored in .my_data1

int main() {
}
```

Vea también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

`check_stack` pragma

12/08/2021 • 2 minutes to read

Indica al compilador que desactive los sondeos de pila si se especifica (o) o que active los sondeos de pila si se especifica `off` `-` `on` `+` (o).

Sintaxis

```
#pragma check_stack( [{ on | off }] )
#pragma check_stack { + | - }
```

Comentarios

Esto pragma tiene efecto en la primera función definida después de que se ve pragma . Las comprobaciones de la pila no forman parte de las macros ni de las funciones insertadas que se generan.

Si no se da un argumento para , la comprobación de la pila vuelve al `check_stack` pragma comportamiento especificado en la línea de comandos. Para obtener más información, vea Opciones del compilador. La interacción de `#pragma check_stack` y la opción se resume en la tabla `/Gs` siguiente.

Utilización de la pragma check_stack

SINTAXIS	¿SE COMPILA CON `/GS` ¿OPCIÓN?	ACCIÓN
`#pragma check_stack( )` o `#pragma check_stack`	Yes	Desactiva la comprobación de la pila para las funciones que la siguen
`#pragma check_stack( )` o `#pragma check_stack`	No	Activa la comprobación de la pila para las funciones que la siguen
`#pragma check_stack(on)` o bien `#pragma check_stack +`	Sí o no	Activa la comprobación de la pila para las funciones que la siguen
`#pragma check_stack(off)` o bien `#pragma check_stack -`	Sí o no	Desactiva la comprobación de la pila para las funciones que la siguen

Vea también

Directivas Pragma y las palabras `__pragma` `_Pragma` clave y

`code_seg` pragma

12/08/2021 • 2 minutes to read

Especifica la sección de texto (segmento) donde las funciones se almacenan en el archivo de objeto (.obj).

Sintaxis

```
#pragma code_seg( [ "section-name" [ , "section-class" ] ] )
#pragma code_seg( { push | pop } [ , identifier ] [ , "section-name" [ , "section-class" ] ] )
```

Parámetros

`push`
(Opcional) Coloca un registro en la pila interna del compilador. Un `push` puede tener un identificador *y* un nombre *de sección.*

`pop`
(Opcional) Quita un registro de la parte superior de la pila interna del compilador. Un `pop` puede tener un identificador *y* un nombre *de sección.* Puede hacer que se puedan obtener varios registros con un `pop` solo comando mediante el *identificador*. *Section-name se* convierte en el nombre de la sección de texto activo después del elemento emergente.

identifier
(Opcional) Cuando se usa con `push` , asigna un nombre al registro en la pila interna del compilador. Cuando se usa con `pop` , la directiva quita los registros de la pila interna hasta que se *quita* el identificador. Si *no* se encuentra el identificador en la pila interna, no se hace nada.

"*section-name*"
(Opcional) Nombre de una sección. Cuando se usa con `pop` , la pila se activa y *section-name* se convierte en el nombre de sección de texto activo.

"*section-class*"
(Opcional) Se omite, pero se incluye por compatibilidad con versiones de Microsoft C++ anteriores a la versión 2.0.

Comentarios

Una *sección* de un archivo de objeto es un bloque de datos con nombre que se carga en la memoria como una unidad. Una *sección de texto* es una sección que contiene código ejecutable. En este artículo, los *términos segment* y *section* tienen el mismo significado.

La directiva indica al compilador que coloque todo el código de objeto posterior de la unidad de traducción en una sección `code_seg` pragma de texto denominada *section-name*. De forma predeterminada, la sección de texto que se usa para las funciones de un archivo de objeto se denomina `.text` . Una `code_seg` pragma directiva sin un parámetro *section-name* restablece el nombre de la sección de texto para el código de objeto subsiguiente a `.text` .

La `code_seg` pragma directiva no controla la ubicación del código de objeto generado para las plantillas con instancias. Tampoco controla el código generado implícitamente por el compilador, como las funciones miembro especiales. Para controlar ese código, se recomienda usar el `__declspec(code_seg(...))` atributo en su lugar. Proporciona control sobre la ubicación de todo el código de objeto, incluido el código generado por el

compilador.

Para obtener una lista de nombres que no se deben usar para crear una sección, vea `/SECTION` .

También puede especificar secciones para los datos inicializados ( ), los datos no inicializados ( ) y `data_seg` las variables `bss_seg` const (`const_seg`).

Puede usar la aplicación DUMPBIN.EXEpara ver los archivos objeto. Las versiones de DUMPBIN para cada arquitectura de destino admitida se incluyen con Visual Studio.

Ejemplo

En este ejemplo se muestra cómo usar **la directiva code_seg** para controlar dónde se coloca el código de pragma objeto:

```
// pragma_directive_code_seg.cpp
void func1() {                  // stored in .text
}

#pragma code_seg(".my_data1")
void func2() {                  // stored in my_data1
}

#pragma code_seg(push, r1, ".my_data2")
void func3() {                  // stored in my_data2
}

#pragma code_seg(pop, r1)      // stored in my_data1
void func4() {
}

int main() {
}
```

Consulte también

`code_seg (__declspec)`
Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

`comment` pragma

16/08/2021 • 3 minutes to read

Inserta un registro de comentario en un archivo objeto o ejecutable.

Sintaxis

```
#pragma comment( comment-type [ , "comment-string" ] )
```

Comentarios

El *tipo de comentario* es uno de los identificadores predefinidos, que se describen a continuación, que especifica el tipo de registro de comentario. La cadena *de comentario opcional* es un literal de cadena que proporciona información adicional para algunos tipos de comentario. Dado *que comment-string* es un literal de cadena, sigue todas las reglas para los literales de cadena en el uso de caracteres de escape, comillas incrustadas (`"`) y concatenación.

compiler

Coloca el nombre y número de versión del compilador en el archivo objeto. El vinculador no tiene en cuenta este registro de comentario. Si proporciona un parámetro *de cadena de* comentario para este tipo de registro, el compilador genera una advertencia.

lib

Inserta un registro de búsqueda de biblioteca en el archivo objeto. Este tipo de comentario debe ir acompañado de un parámetro *de* cadena de comentario que tenga el nombre (y posiblemente la ruta de acceso) de la biblioteca en la que desea que busque el vinculador. El nombre de la biblioteca sigue los registros de búsqueda de biblioteca predeterminados en el archivo de objeto. El vinculador busca esta biblioteca de la misma manera que si la hubiera especificado en la línea de comandos, siempre y cuando la biblioteca no se especifique mediante `/nodefaultlib` . Puede colocar varios registros de búsqueda de biblioteca en el mismo archivo de código fuente. Cada registro aparece en el archivo objeto en el mismo orden en que se encuentra en el archivo de origen.

Si el orden de la biblioteca predeterminada y una biblioteca agregada es importante, la compilación con el modificador impedirá que el nombre de biblioteca predeterminado se coloque `/Zl` en el módulo de objetos. A continuación, se puede usar un segundo comentario pragma para insertar el nombre de la biblioteca predeterminada después de la biblioteca agregada. Las bibliotecas enumeradas con estas directivas aparecerán en el módulo de objetos en el mismo orden en que pragma se encuentran en el código fuente.

vinculador

Coloca una opción del vinculador en el archivo de objeto. Puede utilizar este tipo de comentario para especificar una opción del vinculador en lugar de pasarla a la línea de comandos o de especificarla en el entorno de desarrollo. Por ejemplo, puede especificar la opción /include para forzar la inclusión de un símbolo:

```
#pragma comment(linker, "/include:__mySymbol")
```

Solo están disponibles las siguientes opciones *del vinculador*(tipo de comentario) para pasar al identificador del vinculador:

- `/DEFAULTLIB`

- `/EXPORT`
- `/INCLUDE`
- `/MANIFESTDEPENDENCY`
- `/MERGE`
- `/SECTION`

usuario

Coloca un comentario general en el archivo objeto. El *parámetro comment-string* contiene el texto del comentario. El vinculador no tiene en cuenta este registro de comentario.

Ejemplos

Lo siguiente pragma hace que el vinculador busque emapi. Biblioteca LIB durante la vinculación. El vinculador busca primero en el directorio de trabajo actual y, a continuación, en la ruta de acceso especificada en la variable de entorno LIB.

```
#pragma comment( lib, "emapi" )
```

Lo siguiente pragma hace que el compilador coloque el nombre y el número de versión del compilador en el archivo de objeto:

```
#pragma comment( compiler )
```

Para los comentarios que toman un parámetro *de* cadena de comentario, puede usar una macro en cualquier lugar donde se usaría un literal de cadena, siempre y cuando la macro se expanda a un literal de cadena. También puede concatenar cualquier combinación de literales de cadena y macros que se expandan a literales de cadena. Por ejemplo, la siguiente instrucción es aceptable:

```
#pragma comment( user, "Compiled on " __DATE__ " at " __TIME__ )
```

Vea también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

`component` pragma

16/08/2021 • 2 minutes to read

Controla la colección de información de exploración o información de dependencia desde los archivos de origen.

Sintaxis

```
#pragma component( browser, { on | off } [ , references [ , name ] ] )
#pragma component( minrebuild, { on | off } )
#pragma component( mintypeinfo, { on | off } )
```

Comentarios

Explorador

Puede activar o desactivar la recopilación, así como especificar que determinados nombres se omitan al recopilar la información.

El uso de on o off controla la colección de información de exploración desde el pragma reenvío. Por ejemplo:

```
#pragma component(browser, off)
```

detiene la recopilación de información de examen por parte del compilador.

> **NOTE**
> Para activar la recopilación de información de exploración con este , primero debe pragma habilitarsela información de exploración .

La `references` opción se puede usar con o sin el argumento *name*. El uso de sin nombre activa o desactiva la recopilación de referencias (sin embargo, se sigue recopilando otra información `references` de exploración). Por ejemplo:

```
#pragma component(browser, off, references)
```

detiene la recopilación de información de referencia por parte del compilador.

Usar con nombre e impide que las referencias a `references` `off` *nombre* aparezcan en la ventana de información de exploración. Utilice esta sintaxis para omitir nombres y tipos en los que no esté interesado y reducir el tamaño de los archivos de información de examen. Por ejemplo:

```
#pragma component(browser, off, references, DWORD)
```

omite las referencias a DWORD desde ese momento. Puede volver a activar la recopilación de referencias a DWORD mediante `on` :

```
#pragma component(browser, on, references, DWORD)
```

Esta es la única manera de reanudar la recopilación de referencias al *nombre*; debe activar explícitamente cualquier *nombre* que haya desactivado.

Para evitar que el preprocesador expanda *el* nombre (por ejemplo, expandir NULL a 0), coloque comillas alrededor de él:

```
#pragma component(browser, off, references, "NULL")
```

Recompilación mínima

La característica en desuso `/Gm` (Habilitar recompilación mínima) requiere que el compilador cree y almacene la información de dependencia de clase de C++, que ocupa espacio en disco. Para ahorrar espacio en disco, puede usar siempre que no necesite recopilar información de dependencia, por ejemplo, en archivos `#pragma component( minrebuild, off )` de encabezado invariables. Inserte `#pragma component( minrebuild, on )` después de cambiar las clases para volver a activar la recopilación de dependencias.

Reducir la información de tipos

La `mintypeinfo` opción reduce la información de depuración de la región especificada. El volumen de esta información es considerable, afectando a los archivos .pdb y .obj. No se pueden depurar clases y estructuras en la `mintypeinfo` región. El uso de `mintypeinfo` la opción puede ser útil para evitar la advertencia siguiente:

```
LINK : warning LNK4018: too many type indexes in PDB "filename", discarding subsequent type information
```

Para obtener más información, vea la opción del `/Gm` compilador (Habilitar recompilación mínima).

Vea también

Directivas Pragma y las palabras `__pragma` `_Pragma` clave y

`conform` pragma

17/08/2021 • 2 minutes to read

Específicos de C++

Especifica el comportamiento en tiempo de ejecución de la opción `/Zc:forScope` del compilador.

Sintaxis

```
#pragma conform( name [ , show ] [ { } ] [ [ { } , ] on | off [ , push | pop , identifier [ , { } ] ] on | off ] )
```

Parámetros

Nombre
Especifica el nombre de la opción del compilador que se va a modificar. El único nombre *válido* es `forScope` .

`show`
(Opcional) Hace que la configuración actual *del nombre* (true o false) se muestre mediante un mensaje de advertencia durante la compilación. Por ejemplo: `#pragma conform(forScope, show)` .

`on` , `off`
(Opcional) Establecer *el nombre* en habilita la opción del compilador `on` /Zc:forScope. El valor predeterminado es `off` .

`push`
(Opcional) Inserta el valor actual de *name en* la pila interna del compilador. Si especifica el *identificador*, puede especificar el valor `on` o para el `off` *nombre* que se va a insertar en la pila. Por ejemplo: `#pragma conform(forScope, push, myname, on)` .

`pop`
(Opcional) Establece el valor de *name* en el valor de la parte superior de la pila del compilador interno y, a continuación, hace que la pila se resalte. Si se especifica identifier con , la pila se volverá a mostrar hasta que encuentre el registro con el identificador , que también se mostrará; el valor actual de name en el siguiente registro de la pila se convierte en el nuevo valor para `pop` *name*. Si especifica con `pop` un *identificador que* no está en un registro de la pila, se `pop` omite .

identifier
(Opcional) Se puede incluir con un `push` comando `pop` o . Si *se* usa el identificador, también se puede usar `on` un `off` especificador o .

Ejemplo

```
// pragma_directive_conform.cpp
// compile with: /W1
// C4811 expected
#pragma conform(forScope, show)
#pragma conform(forScope, push, x, on)
#pragma conform(forScope, push, x1, off)
#pragma conform(forScope, push, x2, off)
#pragma conform(forScope, push, x3, off)
#pragma conform(forScope, show)
#pragma conform(forScope, pop, x1)
#pragma conform(forScope, show)

int main() {}
```

Vea también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

`const_seg` pragma

11/08/2021 • 2 minutes to read

Especifica la sección (segmento) donde se almacenan las variables const en el archivo de objeto (.obj).

Sintaxis

```
#pragma const_seg( [ "section-name" [ , "section-class" ] ] )
#pragma const_seg( { push | pop } [ , identifier ] [ , "section-name" [ , "section-class" ] ] )
```

Parámetros

`push`
(Opcional) Coloca un registro en la pila interna del compilador. puede `push` tener un identificador *y* un nombre *de sección.*

`pop`
(Opcional) Quita un registro de la parte superior de la pila interna del compilador. puede `pop` tener un identificador *y* un nombre *de sección.* Puede abrir varios registros con un solo `pop` comando mediante el *identificador*. *Section-name se* convierte en el nombre de la sección const activa después del pop.

identifier
(Opcional) Cuando se usa `push` con , asigna un nombre al registro en la pila interna del compilador. Cuando se usa con `pop` , la directiva quita los registros de la pila interna hasta que se *quita* el identificador. Si *no* se encuentra el identificador en la pila interna, no se hace nada.

"*section-name*"
(Opcional) Nombre de una sección. Cuando se usa `pop` con , la pila se abre y *section-name* se convierte en el nombre de la sección const activa.

"*section-class*"
(Opcional) Se omite, pero se incluye por compatibilidad con versiones de Microsoft C++ anteriores a la versión 2.0.

Comentarios

Una *sección* de un archivo de objeto es un bloque con nombre de datos que se carga en la memoria como una unidad. Una *sección const* es una sección que contiene datos constantes. En este artículo, los *términos segmento* y *sección* tienen el mismo significado.

La directiva indica al compilador que coloque todos los elementos de datos constantes de la unidad de traducción en una `const_seg` pragma sección const denominada *section-name*. La sección predeterminada del archivo de objeto para `const` las variables es `.rdata` . Algunas `const` variables, como las escalares, se inlineen automáticamente en la secuencia de código. El código en forma de subrayado no aparece en `.rdata` . Una `const_seg` pragma directiva sin un parámetro *section-name* restablece el nombre de sección de los elementos `const` de datos posteriores a `.rdata` .

Si define un objeto que requiere inicialización dinámica en `const_seg` , el resultado es un comportamiento indefinido.

Para obtener una lista de nombres que no se deben usar para crear una sección, vea `/SECTION` .

También puede especificar secciones para los datos inicializados (`data_seg`), los datos no inicializados ( ) y las funciones (`bss_seg` `code_seg`).

Puede usar la aplicación DUMPBIN.EXE para ver los archivos de objeto. Las versiones de DUMPBIN para cada arquitectura de destino admitida se incluyen con Visual Studio.

Ejemplo

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

const int i = 7;               // inlined, not stored in .rdata
const char sz1[]= "test1";     // stored in .rdata

#pragma const_seg(".my_data1")
const char sz2[]= "test2";     // stored in .my_data1

#pragma const_seg(push, stack1, ".my_data2")
const char sz3[]= "test3";     // stored in .my_data2

#pragma const_seg(pop, stack1) // pop stack1 from stack
const char sz4[]= "test4";     // stored in .my_data1

int main() {
    using namespace std;
   // const data must be referenced to be put in .obj
   cout << sz1 << endl;
   cout << sz2 << endl;
   cout << sz3 << endl;
   cout << sz4 << endl;
}
```

```
test1
test2
test3
test4
```

Consulte también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

`data_seg` pragma

17/08/2021 • 2 minutes to read

Especifica la sección de datos (segmento) donde las variables inicializadas se almacenan en el archivo de objeto (.obj).

Sintaxis

```
#pragma data_seg( [ "section-name" [ , "section-class" ] ] )
#pragma data_seg( { push | pop } [ , identifier ] [ , "section-name" [ , "section-class" ] ] )
```

Parámetros

`push`
(Opcional) Coloca un registro en la pila interna del compilador. puede `push` tener un identificador *y* un nombre *de sección.*

`pop`
(Opcional) Quita un registro de la parte superior de la pila interna del compilador. puede `pop` tener un identificador *y* un nombre *de sección.* Puede abrir varios registros con un solo `pop` comando mediante el *identificador*. *Section-name se* convierte en el nombre de la sección de datos activos después del pop.

identifier
(Opcional) Cuando se usa `push` con , asigna un nombre al registro en la pila interna del compilador. Cuando se usa con `pop` , quita los registros de la pila interna hasta que se *quita* el identificador. Si *no* se encuentra el identificador en la pila interna, no se hace nada.

identifier permite que se resalte varios registros con un solo `pop` comando.

"section-name"
(Opcional) Nombre de una sección. Cuando se usa con `pop` , la pila se abre y *section-name* se convierte en el nombre de la sección de datos activa.

"section-class"
(Opcional) Se omite, pero se incluye por compatibilidad con versiones de Microsoft C++ anteriores a la versión 2.0.

Comentarios

Una *sección* de un archivo de objeto es un bloque con nombre de datos que se carga en la memoria como una unidad. Una *sección de datos* es una sección que contiene datos inicializados. En este artículo, los *términos segmento* y *sección* tienen el mismo significado.

La sección predeterminada del archivo .obj para las variables inicializadas es `.data` . Las variables que no están inicializadas se consideran inicializadas en cero y se almacenan en `.bss` .

La directiva indica al compilador que coloque todos los elementos de datos inicializados de la unidad de traducción en una sección `data_seg` pragma de datos denominada *section-name*. De forma predeterminada, la sección de datos utilizada para los datos inicializados en un archivo de objeto se denomina `.data` . Las variables que no están inicializadas se consideran inicializadas en cero y se almacenan en `.bss` . Una directiva sin un parámetro section-name restablece el nombre de la sección de `data_seg` pragma datos para los

elementos de datos inicializados posteriores en `.data` .

Los datos `data_seg` asignados mediante no conservan ninguna información sobre su ubicación.

Para obtener una lista de nombres que no se deben usar para crear una sección, vea `/SECTION` .

También puede especificar secciones para las variables const (`const_seg`), los datos no inicializados ( ) y las funciones (`bss_seg` `code_seg`).

Puede usar la aplicación DUMPBIN.EXE para ver los archivos de objeto. Las versiones de DUMPBIN para cada arquitectura de destino admitida se incluyen con Visual Studio.

Ejemplo

```
// pragma_directive_data_seg.cpp
int h = 1;                      // stored in .data
int i = 0;                      // stored in .bss
#pragma data_seg(".my_data1")
int j = 1;                      // stored in .my_data1

#pragma data_seg(push, stack1, ".my_data2")
int l = 2;                      // stored in .my_data2

#pragma data_seg(pop, stack1)   // pop stack1 off the stack
int m = 3;                      // stored in .my_data1

int main() {
}
```

Vea también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

`deprecated` pragma

17/08/2021 • 2 minutes to read

permite indicar que una función, un tipo o cualquier otro identificador ya no se admiten en una versión futura o ya `deprecated` pragma no se deben usar.

> **NOTE**
> Para obtener información sobre el atributo de C++14 e instrucciones sobre cuándo usar ese atributo en lugar del modificador de Microsoft o , vea `[[deprecated]]` `__declspec(deprecated)` Atributos en `deprecated` pragma C++.

Sintaxis

`#pragma deprecated(` *identifier1* [`,` *identifier2* ...] `)`

Comentarios

Cuando el compilador encuentra un identificador especificado por , emite la `deprecated` pragma advertencia del compilador C4995.

Puede desusar nombres de macro. Coloque el nombre de la macro entre comillas; de lo contrario se producirá la expansión de la macro.

Dado que funciona en todos los identificadores correspondientes y no tiene en cuenta las firmas, no es la mejor opción para desusar versiones específicas de funciones `deprecated` pragma sobrecargadas. Cualquier nombre de función que coincida que se entre en el ámbito desencadena la advertencia.

Se recomienda usar el atributo de C++14, siempre `[[deprecated]]` que sea posible, en lugar de `deprecated` pragma . El modificador de declaración específico de Microsoft `__declspec(deprecated)` también es una mejor opción en muchos casos que `deprecated` pragma . El `[[deprecated]]` atributo y el modificador permiten especificar el estado en `__declspec(deprecated)` desuso para determinadas formas de funciones sobrecargadas. La advertencia de diagnóstico solo aparece en las referencias a la función sobrecargada específica a la que se aplica el atributo o modificador.

Ejemplo

```
// pragma_directive_deprecated.cpp
// compile with: /W3
#include <stdio.h>
void func1(void) {
}

void func2(void) {
}

int main() {
   func1();
   func2();
   #pragma deprecated(func1, func2)
   func1();   // C4995
   func2();   // C4995
}
```

En el ejemplo siguiente se muestra cómo desusar una clase:

```
// pragma_directive_deprecated2.cpp
// compile with: /W3
#pragma deprecated(X)
class X {  // C4995
public:
   void f(){}
};

int main() {
   X x;   // C4995
}
```

Vea también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

`detect_mismatch` pragma

16/08/2021 • 2 minutes to read

Inserta un registro en un objeto. El vinculador comprueba estos registros para detectar potenciales discordancias.

Sintaxis

```
#pragma detect_mismatch( "name" , "value" )
```

Comentarios

Al vincular el proyecto, el vinculador produce un error LNK2038 si el proyecto contiene dos objetos que tienen el mismo nombre, pero cada uno tiene un valor *diferente*. Use esta opción pragma para evitar que se vinculen archivos de objetos incoherentes.

Tanto *el nombre* como el *valor* son literales de cadena y cumplen las reglas de los literales de cadena con respecto a los caracteres de escape y la concatenación. Distinguen mayúsculas de minúsculas y no pueden contener una coma, signo igual, comillas ni el **carácter** nulo.

Ejemplo

En este ejemplo se crean dos archivos que tienen números de versión diferentes para la misma etiqueta de versión.

```
// pragma_directive_detect_mismatch_a.cpp
#pragma detect_mismatch("myLib_version", "9")
int main ()
{
   return 0;
}

// pragma_directive_detect_mismatch_b.cpp
#pragma detect_mismatch("myLib_version", "1")
```

Si compila ambos archivos mediante la línea de comandos `cl pragma_directive_detect_mismatch_a.cpp pragma_directive_detect_mismatch_b.cpp` , recibirá el error LNK2038.

Vea también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

`execution_character_set` pragma

16/08/2021 • 2 minutes to read

Especifica el juego de caracteres de ejecución utilizado para los literales de cadena y carácter. Esta directiva no es necesaria para los literales marcados con el `u8` prefijo .

Sintaxis

```
#pragma execution_character_set( "target" )
```

Parámetros

Objetivo
Especifica el juego de caracteres de ejecución de destino. Actualmente, el único conjunto de ejecución de destino admitido es "utf-8".

Comentarios

Esta directiva del compilador está obsoleta a partir de Visual Studio 2015 Update 2. Se recomienda usar las opciones del compilador o junto con el prefijo en literales de cadena y caracteres `/execution-charset:utf-8` `/utf-8` `u8` estrechos que contienen caracteres extendidos. Para obtener más información sobre el `u8` prefijo, vea Cadena y literales de caracteres. Para obtener más información sobre las opciones del compilador, vea `/execution-charset` (Establecer juego de caracteres de ejecución) y (Establecer los juegos de caracteres de origen y ejecutable `/utf-8` en UTF-8).

La directiva indica al compilador que codifique literales de cadena estrecha y caracteres estrechos en el código fuente como `#pragma execution_character_set("utf-8")` UTF-8 en el ejecutable. Esta codificación de salida es independiente de la codificación del archivo de origen utilizada.

De forma predeterminada, el compilador codifica caracteres estrechos y cadenas estrechas mediante la página de códigos actual como el juego de caracteres de ejecución. Esto significa que los caracteres Unicode o DBCS fuera del intervalo de la página de códigos actual se convierten al carácter de reemplazo predeterminado en la salida. Los caracteres Unicode y DBCS se truncan a su byte de orden bajo, que casi nunca es lo que se pretende. Puede especificar la codificación UTF-8 para los literales en el archivo de origen mediante un `u8` prefijo. El compilador pasa estas cadenas codificadas UTF-8 a la salida sin cambios. Los literales de caracteres estrechos con el prefijo u8 deben caber en un byte o se truncan en la salida.

De forma predeterminada, Visual Studio la página de códigos actual como el juego de caracteres de origen que se usa para interpretar el código fuente para la salida. Cuando se lee un archivo, Visual Studio lo interpreta según la página de códigos actual, a menos que se haya establecido la página de códigos del archivo, o a menos que se detecte una marca de orden de bytes (BOM) o caracteres UTF-16 al principio del archivo. No se puede establecer UTF-8 como la página de códigos actual en algunas versiones de Windows. Cuando la detección automática encuentra archivos de origen codificados como UTF-8 sin bom en esas versiones, Visual Studio supone que están codificados mediante la página de códigos actual. Los caracteres del archivo de origen que están fuera del intervalo de la página de códigos especificada o detectada automáticamente pueden provocar advertencias y errores del compilador.

Vea también

Directivas Pragma y las palabras `__pragma` `_Pragma` clave y

`/execution-charset` (Establecer juego de caracteres de ejecución)
`/utf-8` (Establezca los juegos de caracteres de origen y ejecutable en UTF-8)

`fenv_access` pragma

16/08/2021 • 2 minutes to read

Deshabilita ( ) o habilita ( ) optimizaciones que podrían cambiar las pruebas de marca de entorno de punto flotante y `on` `off` los cambios de modo.

Sintaxis

```
#pragma fenv_access ( { on | off } )
```

Comentarios

De forma predeterminada, `fenv_access` es `off` . El compilador supone que el código no tiene acceso al entorno de punto flotante ni lo manipula. Si no se requiere acceso al entorno, el compilador puede hacer más para optimizar el código de punto flotante.

Habilite si el código prueba marcas de estado de punto `fenv_access` flotante, excepciones o establece marcas de modo de control. El compilador deshabilita las optimizaciones de punto flotante, por lo que el código puede acceder al entorno de punto flotante de forma coherente.

La opción de línea de comandos [/fp:strict] habilita automáticamente `fenv_access` . Para obtener más información sobre este y otro comportamiento de punto flotante, vea /fp (Especificar Floating-Point comportamiento).

Existen restricciones en las formas en que se puede usar `fenv_access` pragma en combinación con otras configuraciones de punto flotante:

- No se puede habilitar a `fenv_access` menos que se habilite la semántica precisa. La semántica precisa se puede habilitar mediante `float_control` pragma o mediante las opciones del compilador `/fp:precise` `/fp:strict` o . El compilador tiene como valor predeterminado si no se especifica ninguna otra opción de línea de comandos `/fp:precise` de punto flotante.
- No se puede usar para `float_control` deshabilitar la semántica precisa cuando se establece `fenv_access(on)` .

Los tipos de optimizaciones que están sujetas `fenv_access` a son:

- Eliminación común global de la subexpresión
- Movimiento de código
- Plegamiento constante

Otras directivas de pragma punto flotante incluyen:

- float_control
- fp_contract

Ejemplos

En este ejemplo se `fenv_access` establece en para establecer el registro de control de punto flotante para una precisión de `on` 24 bits:

```
// pragma_directive_fenv_access_x86.cpp
// compile with: /O2 /arch:IA32
// processor: x86
#include <stdio.h>
#include <float.h>
#include <errno.h>
#pragma fenv_access (on)

int main() {
   double z, b = 0.1, t = 0.1;
   unsigned int currentControl;
   errno_t err;

   err = _controlfp_s(&currentControl, _PC_24, _MCW_PC);
   if (err != 0) {
      printf_s("The function _controlfp_s failed!\n");
      return -1;
   }
   z = b * t;
   printf_s ("out=%.15e\n",z);
}
```

```
out=9.999999776482582e-03
```

Si comenta del `#pragma fenv_access (on)` ejemplo anterior, la salida es diferente. Se debe a que el compilador realiza la evaluación en tiempo de compilación, que no usa el modo de control.

```
// pragma_directive_fenv_access_2.cpp
// compile with: /O2 /arch:IA32
#include <stdio.h>
#include <float.h>

int main() {
   double z, b = 0.1, t = 0.1;
   unsigned int currentControl;
   errno_t err;

   err = _controlfp_s(&currentControl, _PC_24, _MCW_PC);
   if (err != 0) {
      printf_s("The function _controlfp_s failed!\n");
      return -1;
   }
   z = b * t;
   printf_s ("out=%.15e\n",z);
}
```

```
out=1.000000000000000e-02
```

Vea también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

`float_control` pragma

12/08/2021 • 2 minutes to read

Especifica el comportamiento de punto flotante de una función.

Sintaxis

```
#pragma float_control
#pragma float_control( precise, { on | off } [ , push ] )
#pragma float_control( except, { on | off } [ , push ] )
#pragma float_control( { push | pop } )
```

Opciones

`precise` , `on` | `off` , `push`
Especifica si se debe habilitar (`on`) o deshabilitar ( ) semántica precisa de punto `off` flotante. Para obtener información sobre las `/fp:precise` diferencias con la opción del compilador, vea la sección Comentarios. El `push` token opcional inserta la configuración actual de en `float_control` la pila interna del compilador.

`except` , `on` | `off` , `push`
Especifica si se debe habilitar (`on`) o deshabilitar ( ) semántica de `off` excepción de punto flotante. El `push` token opcional inserta la configuración actual de en `float_control` la pila interna del compilador.

`except` solo se puede establecer en `on` cuando también se establece en `precise` `on` .

`push`
Inserta la configuración actual `float_control` en la pila interna del compilador.

`pop`
Quita la configuración de la parte superior de la pila interna del compilador y la `float_control` convierte en la nueva `float_control` configuración.

Comentarios

no `float_control` pragma tiene el mismo comportamiento que la opción `/fp` del compilador. Solo `float_control` pragma rige parte del comportamiento de punto flotante. Debe combinarse con las `fp_contract` directivas y para volver a crear las opciones del `fenv_access` pragma `/fp` compilador. En la tabla siguiente se muestra la configuración pragma equivalente para cada opción del compilador:

OPCIÓN	`FLOAT_CONTROL(PRECISE, *)`	`FLOAT_CONTROL(EXCEPT, *)`	`FP_CONTRACT(*)`	`FENV_ACCESS(*)`
`/fp:strict`	`on`	`on`	`off`	`on`
`/fp:precise`	`on`	`off`	`on`	`off`
`/fp:fast`	`off`	`off`	`on`	`off`

En otras palabras, puede que necesite usar varias directivas en combinación para emular las opciones de línea pragma de comandos , y `/fp:fast` `/fp:precise` `/fp:strict` .

Existen restricciones en las formas en que puede usar las directivas de punto flotante y `float_control` `fenv_access` en pragma combinación:

- Solo puede usar para `float_control` establecer en si está habilitada la semántica `except` `on` precisa. La semántica precisa se puede habilitar mediante `float_control` pragma o mediante las opciones del compilador `/fp:precise` `/fp:strict` o .
- No se puede usar para desactivar cuando la semántica de excepciones `float_control` `precise` está habilitada, ya sea mediante una `float_control` pragma opción del compilador `/fp:except` o .
- No se puede habilitar a `fenv_access` menos que se habilite la semántica precisa, ya sea mediante una opción del compilador o `float_control` pragma .
- No se puede usar `float_control` para desactivar cuando está `precise` `fenv_access` habilitado.

Estas restricciones significan que el orden de algunas directivas pragma de punto flotante es significativo. Para pasar de un modelo rápido a un modelo estricto mediante pragma directivas, use el código siguiente:

```
#pragma float_control(precise, on)  // enable precise semantics
#pragma fenv_access(on)             // enable environment sensitivity
#pragma float_control(except, on)   // enable exception semantics
#pragma fp_contract(off)            // disable contractions
```

Para pasar de un modelo estricto a un modelo rápido mediante `float_control` pragma , use el código siguiente:

```
#pragma float_control(except, off)  // disable exception semantics
#pragma fenv_access(off)            // disable environment sensitivity
#pragma float_control(precise, off) // disable precise semantics
#pragma fp_contract(on)             // enable contractions
```

Si no se especifica ninguna opción, `float_control` no tiene ningún efecto.

Ejemplo

En el ejemplo siguiente se muestra cómo detectar una excepción de punto flotante de desbordamiento mediante pragma `float_control` .

```
// pragma_directive_float_control.cpp
// compile with: /EHa
#include <stdio.h>
#include <float.h>

double func( ) {
   return 1.1e75;
}

#pragma float_control (except, on)

int main( ) {
   float u[1];
   unsigned int currentControl;
   errno_t err;

   err = _controlfp_s(&currentControl, ~_EM_OVERFLOW, _MCW_EM);
   if (err != 0)
      printf_s("_controlfp_s failed!\n");

   try  {
      u[0] = func();
      printf_s ("Fail");
      return(1);
   }

   catch (...)  {
      printf_s ("Pass");
      return(0);
   }
}
```

```
Pass
```

Consulte también

Directivas Pragma y las palabras `__pragma` `_Pragma` clave y

`fenv_access` pragma

`fp_contract` pragma

`fp_contract` pragma

13/08/2021 • 2 minutes to read

Determina si se produce la contracción de punto flotante. Una contracción de punto flotante es una instrucción como FMA (Fused-Multiply-Add) que combina dos operaciones de punto flotante independientes en una sola instrucción. El uso de estas instrucciones puede afectar a la precisión de punto flotante, ya que, en lugar de redondear después de cada operación, el procesador puede redondear solo una vez después de ambas operaciones.

Sintaxis

```
#pragma fp_contract ( { on | off } )
```

Comentarios

De forma predeterminada, `fp_contract` es `on` . Esto indica al compilador que use instrucciones de contracción de punto flotante siempre que sea posible. Establezca `fp_contract` en para conservar las instrucciones `off` individuales de punto flotante.

Para obtener más información sobre el comportamiento de punto flotante, vea `/fp` (Especificar comportamiento de punto flotante).

Otras directivas de pragma punto flotante incluyen:

- `fenv_access`
- `float_control`

Ejemplo

El código generado a partir de este ejemplo no usa una instrucción fused-multiply-add incluso cuando está disponible en el procesador de destino. Si marca como comentario , el código generado puede usar una instrucción `#pragma fp_contract (off)` fused-multiply-add si está disponible.

```
// pragma_directive_fp_contract.cpp
// on x86 and x64 compile with: /O2 /fp:fast /arch:AVX2
// other platforms compile with: /O2

#include <stdio.h>

// remove the following line to enable FP contractions
#pragma fp_contract (off)

int main() {
   double z, b, t;

   for (int i = 0; i < 10; i++) {
      b = i * 5.5;
      t = i * 56.025;

      z = t * i + b;
      printf("out = %.15e\n", z);
   }
}
```

```
out = 0.000000000000000e+00
out = 6.152500000000000e+01
out = 2.351000000000000e+02
out = 5.207249999999999e+02
out = 9.184000000000000e+02
out = 1.428125000000000e+03
out = 2.049900000000000e+03
out = 2.783725000000000e+03
out = 3.629600000000000e+03
out = 4.587525000000000e+03
```

Vea también

Directivas Pragma y las palabras `__pragma` `_Pragma` clave y

`function` pragma

12/08/2021 • 2 minutes to read

Indica al compilador que genere llamadas a funciones especificadas en la lista de argumentos de , en lugar de pragma inlineing them.

Sintaxis

```
#pragma function( function1 [ , function2 ... ] )
```

Comentarios

Las funciones intrínsecas normalmente se generan como código en línea, no como llamadas de función. Si usa `intrinsic` pragma la opción del compilador o para decir al compilador que genere funciones intrínsecas, puede usar para forzar explícitamente `/Oi` una llamada de `function` pragma función. Una vez `function` pragma que se ve , tiene efecto en la primera definición de función que contiene una función intrínseca especificada. El efecto continúa hasta el final del archivo de origen o hasta la apariencia de una que `intrinsic` pragma especifica la misma función intrínseca. Solo puede usar el `function` pragma fuera de una función, en el nivel global.

Para obtener listas de las funciones `intrinsic` pragma que tienen formatos intrínsecos, vea .

Ejemplo

```
// pragma_directive_function.cpp
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// use intrinsic forms of memset and strlen
#pragma intrinsic(memset, strlen)

// Find first word break in string, and set remaining
// chars in string to specified char value.
char *set_str_after_word(char *string, char ch) {
   int i;
   int len = strlen(string);  /* NOTE: uses intrinsic for strlen */

   for(i = 0; i < len; i++) {
      if (isspace(*(string + i)))
         break;
   }

   for(; i < len; i++)
      *(string + i) = ch;

   return string;
}

// do not use strlen intrinsic
#pragma function(strlen)

// Set all chars in string to specified char value.
char *set_str(char *string, char ch) {
   // Uses intrinsic for memset, but calls strlen library function
   return (char *) memset(string, ch, strlen(string));
}

int main() {
   char *str = (char *) malloc(20 * sizeof(char));

   strcpy_s(str, sizeof("Now is the time"), "Now is the time");
   printf("str is '%s'\n", set_str_after_word(str, '*'));
   printf("str is '%s'\n", set_str(str, '!'));
}
```

```
str is 'Now**********'
str is '!!!!!!!!!!!!!!!'
```

Consulte también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

`hdrstop` pragma

14/08/2021 • 2 minutes to read

Proporciona más control sobre los nombres de archivo de precompilación y sobre la ubicación en la que se guarda el estado de compilación.

Sintaxis

```
#pragma hdrstop [ ( "filename" ) ]
```

Comentarios

El *nombre* de archivo es el nombre del archivo de encabezado precompilado que se va a usar o crear (en función de si `/Yu` se especifica o `/Yc`). Si *filename* no contiene una especificación de ruta de acceso, se supone que el archivo de encabezado precompilado está en el mismo directorio que el archivo de origen.

Si un archivo de C o C++ contiene cuando se compila con , el compilador guarda el estado de la compilación hasta la `hdrstop` pragma ubicación de `/Yc` pragma . El estado compilado de cualquier código que sigue a pragma no se guarda.

Use *filename* para dar nombre al archivo de encabezado precompilado en el que se guarda el estado compilado. Un espacio entre `hdrstop` y *filename es* opcional. El nombre de archivo especificado en es una cadena y está sujeto a las `hdrstop` pragma restricciones de cualquier cadena de C o C++. En concreto, debe incluirlo entre comillas y usar el carácter de escape (barra diagonal inversa, `\`) para especificar los nombres de directorio. Por ejemplo:

```
#pragma hdrstop( "c:\\projects\\include\\myinc.pch" )
```

El nombre del archivo de encabezado precompilado se determina según las reglas siguientes, en orden de prioridad:

1. Argumento de la opción `/Fp` del compilador
2. Argumento *de nombre de* archivo para `#pragma hdrstop`
3. Nombre base del archivo de origen con una extensión PCH

Si ninguna de las opciones y o especifica un nombre de archivo, el nombre base del archivo de origen se usa como nombre base del archivo de encabezado `/Yc` `/Yu` `hdrstop` pragma precompilado.

Puede usar también comandos de preprocesamiento para realizar la sustitución de macros del modo siguiente:

```
#define INCLUDE_PATH "c:\\progra~1\\devstsu~1\\vc\\include\\"
#define PCH_FNAME "PROG.PCH"
.
.
.
#pragma hdrstop( INCLUDE_PATH PCH_FNAME )
```

Las reglas siguientes rigen dónde `hdrstop` pragma se puede colocar :

- Debe aparecer fuera de cualquier declaración o definición de datos o función.
- Se debe especificar en el archivo de código fuente, no en un archivo de encabezado.

Ejemplo

```
#include <windows.h>                // Include several files
#include "myhdr.h"

__inline Disp( char *szToDisplay )  // Define an inline function
{
    // ...                          // Some code to display string
}
#pragma hdrstop
```

En este ejemplo, aparece después de que se hayan incluido dos archivos y se haya definido `hdrstop` pragma una función insertado. Esta ubicación podría parecer, al principio, una ubicación impar para pragma . Tenga en cuenta, sin embargo, que el uso de las opciones de precompilación manual, y , con permite precompilar archivos de código fuente completos o incluso código `/Yc` `/Yu` en `hdrstop` pragma línea. El compilador de Microsoft no le limita a precompilar solo declaraciones de datos.

Vea también

Directivas Pragma y las palabras `__pragma` `_Pragma` clave y

`include_alias` pragma

12/08/2021 • 2 minutes to read

Especifica que, *cuando alias_filename* se encuentra en una directiva, el compilador `#include` *actual_filename* en su lugar.

Sintaxis

```
#pragma include_alias( "alias_filename" , " "actual_filename" )
#pragma include_alias( <alias_filename> , <actual_filename> )
```

Comentarios

La directiva permite sustituir los archivos que tienen nombres o rutas de acceso diferentes por los `include_alias` pragma nombres de archivo incluidos por los archivos de origen. Por ejemplo, algunos sistemas de archivos permiten nombres de archivo de encabezado más largos que el límite del sistema de archivos FAT 8.3. El compilador no puede truncar simplemente los nombres más largos a 8.3, porque los primeros ocho caracteres de los nombres de archivo de encabezado más largos pueden no ser únicos. Cada vez que el compilador ve la *alias_filename* en una directiva, sustituye el nombre `#include` actual_filename en su *lugar.* A continuación, carga el *actual_filename* de encabezado. Debe pragma aparecer antes de las directivas `#include` correspondientes. Por ejemplo:

```
// First eight characters of these two files not unique.
#pragma include_alias( "AppleSystemHeaderQuickdraw.h", "quickdra.h" )
#pragma include_alias( "AppleSystemHeaderFruit.h", "fruit.h" )

#pragma include_alias( "GraphicsMenu.h", "gramenu.h" )

#include "AppleSystemHeaderQuickdraw.h"
#include "AppleSystemHeaderFruit.h"
#include "GraphicsMenu.h"
```

El alias que se buscará debe coincidir exactamente con la especificación. Las mayúsculas y minúsculas, la ortografía y el uso de comillas dobles o corchetes angulares deben coincidir. hace `include_alias` pragma que la cadena simple coincida en los nombres de archivo. No se realiza ninguna otra validación de nombre de archivo. Por ejemplo, dadas las siguientes directivas,

```
#pragma include_alias("mymath.h", "math.h")
#include "./mymath.h"
#include "sys/mymath.h"
```

no se realiza ninguna sustitución de alias, ya que las cadenas del archivo de encabezado no coinciden exactamente. Además, los nombres de archivo de encabezado usados como argumentos para las opciones del compilador y , o `/Yu` `/Yc` , no se `hdrstop` pragma sustituyen. Por ejemplo, si el archivo de código fuente contiene la siguiente directiva,

```
#include <AppleSystemHeaderStop.h>
```

la opción del compilador correspondiente debe ser

`/YcAppleSystemHeaderStop.h`

Puede usar para asignar `include_alias` pragma cualquier nombre de archivo de encabezado a otro. Por ejemplo:

```
#pragma include_alias( "api.h", "c:\version1.0\api.h" )
#pragma include_alias( <stdio.h>, <newstdio.h> )
#include "api.h"
#include <stdio.h>
```

No mezcle nombres de archivo entre comillas dobles con nombres de archivo entre corchetes angulares. Por ejemplo, dadas las dos directivas anteriores, el compilador no realiza `#pragma include_alias` ninguna sustitución en las directivas `#include` siguientes:

```
#include <api.h>
#include "stdio.h"
```

Además, la directiva siguiente genera un error:

```
#pragma include_alias(<header.h>, "header.h")  // Error
```

El nombre de archivo notificado en los mensajes de error, o como el valor de la macro predefinida, es el nombre del archivo una vez realizada `__FILE__` la sustitución. Por ejemplo, vea la salida después de las siguientes directivas:

```
#pragma include_alias( "VERYLONGFILENAME.H", "myfile.h" )
#include "VERYLONGFILENAME.H"
```

Un error en *`VERYLONGFILENAME.H`* genera el siguiente mensaje de error:

```
myfile.h(15) : error C2059 : syntax error
```

Tenga en cuenta también que no se admite la transitividad. Dadas las siguientes directivas,

```
#pragma include_alias( "one.h", "two.h" )
#pragma include_alias( "two.h", "three.h" )
#include "one.h"
```

el compilador busca el archivo en *`two.h`* lugar de *`three.h`* .

Consulte también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

`init_seg` pragma

17/08/2021 • 3 minutes to read

Específicos de C++

Especifica una palabra clave o una sección de código que afecta al orden en que se ejecuta el código de inicio.

Sintaxis

```
#pragma init_seg( { compiler | lib | user | "section-name" [ , func-name ] } )
```

Comentarios

Los *términos segmento y sección* tienen el mismo significado en este artículo.

Dado que a veces se requiere código para inicializar objetos estáticos globales, debe especificar cuándo se deben construir los objetos. En concreto, es importante usar en bibliotecas de `init_seg` pragma vínculos dinámicos (DLL) o en bibliotecas que requieren inicialización.

Las opciones para `init_seg` pragma son:

`compiler`
Reservada para la inicialización de la biblioteca en tiempo de ejecución de C de Microsoft. Los objetos de este grupo se construyen en primer lugar.

`lib`
Disponible para las inicializaciones de los proveedores de bibliotecas de clases de terceros. Los objetos de este grupo se construyen después de los marcados como `compiler` , pero antes que cualquier otro.

`user`
Disponible para cualquier usuario. Los objetos de este grupo se construyen en último lugar.

section-name
Permite la especificación explícita de la sección de inicialización. Los objetos de un nombre de *sección especificado por el usuario* no se construyen implícitamente. Sin embargo, sus direcciones se colocan en la sección denominada *por section-name*.

El *nombre de sección* que asigne contendrá punteros a funciones auxiliares que construirán los objetos globales declarados después de en ese pragma módulo.

Para obtener una lista de nombres que no debe usar al crear una sección, vea `/SECTION` .

func-name
Especifica la función que se va a llamar en lugar de `atexit` cuando termine el programa. Esta función auxiliar también llama `atexit` a con un puntero al destructor para el objeto global. Si especifica un identificador de función en pragma el del formulario,

```
int __cdecl myexit (void (__cdecl *pf)(void))
```

se llamará a su función en lugar de a `atexit` de la biblioteca en tiempo de ejecución de C. Permite crear una lista de los destructores a los que llamar cuando esté listo para destruir los objetos.

Si necesita diferir la inicialización (por ejemplo, en una DLL), puede elegir especificar el nombre de sección explícitamente. A continuación, el código debe llamar a los constructores de cada objeto estático.

No se incluyen comillas alrededor del identificador del sustituto de `atexit`.

Los objetos se seguirán colocando en las secciones definidas por las `XXX_seg` pragma otras directivas.

El tiempo de ejecución de C no inicializa automáticamente los objetos declarados en el módulo. El código tiene que realizar la inicialización.

De forma predeterminada, las secciones `init_seg` son de solo lectura. Si el nombre de la sección es , el compilador cambia silenciosamente el atributo para que sea de solo lectura, aunque esté marcado como `.CRT` de lectura, escriba.

No se puede especificar `init_seg` más de una vez en una unidad de traducción.

Incluso si el objeto no tiene un constructor definido por el usuario, uno definido explícitamente en el código, el compilador puede generar uno automáticamente. Por ejemplo, puede crear uno para enlazar punteros de tabla virtual. Cuando sea necesario, el código llama al constructor generado por el compilador.

Ejemplo

```
// pragma_directive_init_seg.cpp
#include <stdio.h>
#pragma warning(disable : 4075)

typedef void (__cdecl *PF)(void);
int cxpf = 0;   // number of destructors we need to call
PF pfx[200];    // pointers to destructors.

int myexit (PF pf) {
   pfx[cxpf++] = pf;
   return 0;
}

struct A {
   A() { puts("A()"); }
   ~A() { puts("~A()"); }
};

// ctor & dtor called by CRT startup code
// because this is before the pragma init_seg
A aaaa;

// The order here is important.
// Section names must be 8 characters or less.
// The sections with the same name before the $
// are merged into one section. The order that
// they are merged is determined by sorting
// the characters after the $.
// InitSegStart and InitSegEnd are used to set
// boundaries so we can find the real functions
// that we need to call for initialization.

#pragma section(".mine$a", read)
__declspec(allocate(".mine$a")) const PF InitSegStart = (PF)1;

#pragma section(".mine$z",read)
__declspec(allocate(".mine$z")) const PF InitSegEnd = (PF)1;

// The comparison for 0 is important.
// For now, each section is 256 bytes. When they
// are merged, they are padded with zeros. You
// can't depend on the section being 256 bytes, but
```

```
// you can depend on it being padded with zeros.

void InitializeObjects () {
   const PF *x = &InitSegStart;
   for (++x ; x < &InitSegEnd ; ++x)
      if (*x) (*x)();
}

void DestroyObjects () {
   while (cxpf>0) {
      --cxpf;
      (pfx[cxpf])();
   }
}

// by default, goes into a read only section
#pragma init_seg(".mine$m", myexit)

A bbbb;
A cccc;

int main () {
   InitializeObjects();
   DestroyObjects();
}
```

```
A()
A()
A()
~A()
~A()
~A()
```

Vea también

Directivas Pragma y las palabras `__pragma` `_Pragma` clave y

`inline_depth` pragma

12/08/2021 • 2 minutes to read

Especifica la profundidad de búsqueda heurística insertda. Las funciones con una profundidad en el gráfico de llamadas mayor que el valor especificado no están en línea.

Sintaxis

```
#pragma inline_depth( [ n ] )
```

Comentarios

Esto pragma controla la inlineación de funciones marcadas `inline` y , o `__inline` inlineed automáticamente en la opción `/Ob` del compilador. Para obtener más información, vea `/Ob` (Expansión de función inserta).

n puede ser un valor entre 0 y 255, donde 255 significa una profundidad ilimitada en el gráfico de llamadas. Un valor de 0 impide la expansión en línea. Cuando *no* se especifica n, se usa el valor predeterminado 254.

controla el número de veces que se puede expandir una serie de llamadas `inline_depth` pragma de función. Por ejemplo, suponga que la profundidad en línea es 4. Si A llama a B y B, a continuación, llama a C, las tres llamadas se expanden en línea. Sin embargo, si la expansión de profundidad insertada más cercana es 2, solo se expanden A y B, y C permanece como una llamada de función.

Para usar pragma este , debe establecer la opción del compilador en `/Ob` 1 o superior. El conjunto de profundidad que usa pragma este tiene efecto en la primera llamada de función después de pragma .

La profundidad en línea se puede reducir durante la expansión, pero no aumentar. Si la profundidad en línea es 6 y, durante la expansión, el preprocesador encuentra un con un valor de `inline_depth` pragma 8, la profundidad permanece en 6.

no `inline_depth` pragma tiene ningún efecto en las funciones marcadas con `__forceinline` .

> **NOTE**
> Las funciones recursivas pueden sustituirse alineadas con una profundidad máxima de 16 llamadas.

Vea también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y `inline_recursion`

`inline_recursion` pragma

14/08/2021 • 2 minutes to read

Controla la expansión en línea de llamadas de función directas o mutuamente recursivas.

Sintaxis

```
#pragma inline_recursion( [ { on | off } ] )
```

Comentarios

Use esta pragma opción para controlar las funciones marcadas como y o que el compilador `inline` `__inline` expande automáticamente en la opción `/Ob2` . El uso de esto pragma requiere un valor de opción del compilador de `/Ob` 1 o 2. El estado predeterminado de `inline_recursion` está desactivado. Esto tiene efecto en la primera llamada de función después de que se ve y pragma no afecta a la pragma definición de la función.

controla `inline_recursion` pragma cómo se expanden las funciones recursivas. Si está desactivada y una función insertada se llama a sí misma, directa o indirectamente, la función se `inline_recursion` expande solo una vez. Si está en, la función se expande varias veces hasta que alcanza el valor establecido con , el valor predeterminado para las funciones `inline_recursion` `inline_depth` pragma recursivas definidas por o un límite `inline_depth` pragma de capacidad.

Vea también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y
`inline_depth`
`/Ob` (Expansión de función inserta)

`intrinsic` pragma

14/08/2021 • 2 minutes to read

Especifica que las llamadas a funciones especificadas en la lista de argumentos de son pragma intrínsecas.

Sintaxis

```
#pragma intrinsic( function_1 [ , function_2 ... ] )
```

Comentarios

indica `intrinsic` pragma al compilador que una función tiene un comportamiento conocido. El compilador puede llamar a la función y no reemplazar la llamada a función con instrucciones alineadas si ello mejora el rendimiento.

A continuación se enumeran las funciones de biblioteca con formas intrínsecas. Una vez `intrinsic` pragma que se ve , tiene efecto en la primera definición de función que contiene una función intrínseca especificada. El efecto continúa hasta el final del archivo de origen o hasta la apariencia de un que `function` pragma especifica la misma función intrínseca. solo `intrinsic` pragma se puede usar fuera de una definición de función, en el nivel global.

Las funciones siguientes tienen formularios intrínsecos y los formularios intrínsecos se usan cuando se especifica `/Oi` :

`abs`
`_disable`
`_enable`
`fabs`
`_inp`
`_inpw` \

`labs`
`_lrotl`
`_lrotr`
`memcmp`
`memcpy` \

`memset`
`_outp`
`_outpw`
`_rotl`
`_rotr` \

`strcat`
`strcmp`
`strcpy`
`strlen`
`_strset` \

Los programas que usan funciones intrínsecas son más rápidos porque no tienen la sobrecarga de las llamadas

de función. Sin embargo, pueden ser mayores debido al código adicional generado.

Ejemplo específico de x86

Los intrínsecos y generan instrucciones en modo kernel para deshabilitar o habilitar interrupciones, y `_disable` `_enable` podrían ser útiles en los controladores en modo kernel.

Compile el código siguiente desde la línea de comandos con y mire para ver que se convierten en instrucciones `cl -c -FAs sample.c` *`sample.asm`* x86 CLI y LANS:

```
// pragma_directive_intrinsic.cpp
// processor: x86
#include <dos.h>   // definitions for _disable, _enable
#pragma intrinsic(_disable)
#pragma intrinsic(_enable)
void f1(void) {
   _disable();
   // do some work here that should not be interrupted
   _enable();
}
int main() {
}
```

Funciones intrínsecas de punto flotante

Estas funciones de punto flotante no tienen formas intrínsecas verdaderas. En su lugar, tienen versiones que pasan argumentos directamente al chip de punto flotante, en lugar de insertarlos en la pila:

`acos`
`asin` \

`cosh`
`fmod` \

`pow`
`sinh` \

`tanh` \

Estas funciones de punto flotante tienen formas intrínsecas verdaderas al especificar y (o cualquier opción que `/Oi` incluya : , y `/fp:fast` `/Oi` `/Ox` `/O1` `/O2`):

`atan`
`atan2`
`cos` \

`exp`
`log` \

`log10`
`sin` \

`sqrt`
`tan` \

Puede usar o `/fp:strict` para `/Za` invalidar la generación de verdaderas opciones intrínsecas de punto flotante. En este caso, las funciones se generan como rutinas de biblioteca que pasan los argumentos directamente al chip de punto flotante, en lugar de insertarlos en la pila del programa.

Vea `#pragma function` para obtener información y un ejemplo sobre cómo habilitar y deshabilitar intrínsecos para un bloque de texto de origen.

Vea también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y Intrínsecos del controlador

`loop` pragma

17/08/2021 • 2 minutes to read

Controla cómo el paralelizador automático debe considerar el código de bucle o excluye un bucle de la consideración del vectorizador automático.

Sintaxis

```
#pragma loop( hint_parallel( n ) )
#pragma loop( no_vector )
#pragma loop( ivdep )
```

Parámetros

`hint_parallel( n )`
Sugerencia al compilador de que este bucle se debe paralelizar entre *n* subprocesos, donde *n* es un literal entero positivo o cero. Si *n* es cero, se usa el número máximo de subprocesos en tiempo de ejecución. Es una sugerencia para el compilador, no un comando. No hay ninguna garantía de que el bucle se va a paralelizar. Si el bucle tiene dependencias de datos o problemas estructurales, no se paralelizará. Por ejemplo, no se paraleliza si almacena en un escalar que se usa más allá del cuerpo del bucle.

El compilador omite esta opción a menos que `/Qpar` se especifique el modificador del compilador.

`no_vector`
De forma predeterminada, el vectorizador automático intenta vectorizar todos los bucles que evalúa pueden beneficiarse de él. Especifique esto pragma para deshabilitar el vectorizador automático para el bucle siguiente.

`ivdep`
Sugerencia al compilador para omitir las dependencias vectoriales de este bucle.

Comentarios

Para usar `loop` pragma , colómelo inmediatamente antes, no en, de una definición de bucle. tiene pragma efecto para el ámbito del bucle que lo sigue. Puede aplicar varias directivas a un bucle, en cualquier orden, pero debe declarar cada una pragma de ellas en una instrucción pragma independiente.

Vea también

Paralelización automática y vectorización automática
Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

`make_public` pragma

16/08/2021 • 2 minutes to read

Indica que un tipo nativo debe tener accesibilidad pública de ensamblado.

Sintaxis

```
#pragma make_public( type )
```

Parámetros

type
Nombre del tipo que desea que tenga accesibilidad de ensamblado público.

Comentarios

`make_public` es útil cuando el tipo nativo al que desea hacer referencia es de un archivo de encabezado que no se puede cambiar. Si desea usar el tipo nativo en la firma de una función pública en un tipo con visibilidad de ensamblado público, el tipo nativo también debe tener accesibilidad de ensamblado público o el compilador emitirá una advertencia.

`make_public` debe especificarse en el ámbito global. Solo está en vigor desde el punto en el que se declara hasta el final del archivo de código fuente.

El tipo nativo puede ser implícita o explícitamente privado. Para obtener más información, vea Visibilidad de tipos.

Ejemplos

El ejemplo siguiente es el contenido de un archivo de encabezado que contiene las definiciones de dos estructuras nativas.

```
// make_public_pragma.h
struct Native_Struct_1 { int i; };
struct Native_Struct_2 { int i; };
```

El ejemplo de código siguiente consume el archivo de encabezado. Muestra que, a menos que marque explícitamente los structs nativos como públicos mediante , el compilador generará una advertencia cuando intente usar los structs nativos en la firma de la función pública en un tipo administrado `make_public` público.

```
// make_public_pragma.cpp
// compile with: /c /clr /W1
#pragma warning (default : 4692)
#include "make_public_pragma.h"
#pragma make_public(Native_Struct_1)

public ref struct A {
   void Test(Native_Struct_1 u) {u.i = 0;}   // OK
   void Test(Native_Struct_2 u) {u.i = 0;}   // C4692
};
```

Vea también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

`managed` y `unmanaged` pragma

17/08/2021 • 2 minutes to read

Habilite el control de nivel de función para compilar funciones como administradas o no administradas.

Sintaxis

```
#pragma managed
#pragma unmanaged
#pragma managed( [ push, ] { on | off } )
#pragma managed(pop)
```

Comentarios

La `/clr` opción del compilador proporciona control de nivel de módulo para compilar funciones como administradas o no administradas.

Se compila una función no administrada para la plataforma nativa. Common Language Runtime pasará la ejecución de esa parte del programa a la plataforma nativa.

Las funciones se compilan como administradas de forma predeterminada cuando `/clr` se usa .

Al aplicar o `managed` `unmanaged` pragma :

- Agregue la pragma función anterior, pero no dentro de un cuerpo de función.
- Agregue las pragma instrucciones `#include` after. No lo use antes que ninguna `#include` instrucción.

El compilador omite y `managed` `unmanaged` pragma si no se usa en `/clr` la compilación.

Cuando se crea una instancia de una función de plantilla, el estado cuando se define la plantilla determina si pragma está administrada o no administrada.

Para obtener más información, vea Inicialización de ensamblados mixtos.

Ejemplo

```
// pragma_directives_managed_unmanaged.cpp
// compile with: /clr
#include <stdio.h>

// func1 is managed
void func1() {
   System::Console::WriteLine("In managed function.");
}

// #pragma unmanaged
// push managed state on to stack and set unmanaged state
#pragma managed(push, off)

// func2 is unmanaged
void func2() {
   printf("In unmanaged function.\n");
}

// #pragma managed
#pragma managed(pop)

// main is managed
int main() {
   func1();
   func2();
}
```

```
In managed function.
In unmanaged function.
```

Vea también

Directivas Pragma y las palabras `__pragma` `_Pragma` clave y

`message` pragma

16/08/2021 • 2 minutes to read

Envía un literal de cadena al resultado estándar sin finalizar la compilación.

Sintaxis

```
#pragma message( message-string )
```

Comentarios

Un uso típico de es `message` pragma mostrar mensajes informativos en tiempo de compilación.

El *parámetro message-string* puede ser una macro que se expande a un literal de cadena y puede concatenar dichas macros con literales de cadena en cualquier combinación.

Si usa una macro predefinida en `message` pragma , la macro debe devolver una cadena. De lo contrario, tendrá que convertir la salida de la macro en una cadena.

El fragmento de código siguiente usa `message` pragma para mostrar los mensajes durante la compilación:

```
// pragma_directives_message1.cpp
// compile with: /LD
#if _M_IX86 >= 500
#pragma message("_M_IX86 >= 500")
#endif

#pragma message("")

#pragma message( "Compiling " __FILE__ )
#pragma message( "Last modified on " __TIMESTAMP__ )

#pragma message("")

// with line number
#define STRING2(x) #x
#define STRING(x) STRING2(x)

#pragma message (__FILE__ "[" STRING(__LINE__) "]: test")

#pragma message("")
```

Vea también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

`omp` pragma

13/08/2021 • 2 minutes to read

Toma una o más directivas de OpenMP, junto con cualquier cláusula directiva opcional.

Sintaxis

> ** `#pragma omp` ** *directiva*

Comentarios

Para obtener más información, vea Directivas de OpenMP.

Vea también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

`once` pragma

17/08/2021 • 2 minutes to read

Especifica que el compilador incluye el archivo de encabezado solo una vez, al compilar un archivo de código fuente.

Sintaxis

```
#pragma once
```

Comentarios

El uso de puede reducir los tiempos de compilación, ya que el compilador no volverá a abrir y leer el archivo después del primero del archivo `#pragma once` en la unidad de `#include` traducción. Se denomina optimización *de varios elementos de incluyeción.* Tiene un efecto similar al lenguaje *de* protección de inclusión, que usa definiciones de macro de preprocesador para evitar varias inclusiones del contenido del archivo. También ayuda a evitar infracciones de una regla de definición: el requisito de que todas las plantillas, tipos, funciones y objetos no tengan más de una definición en el código.

Por ejemplo:

```
// header.h
#pragma once
// Code placed here is included only once per translation unit
```

Se recomienda la directiva `#pragma once` para el nuevo código, ya que no contamina el espacio de nombres global con un símbolo de preprocesador. Requiere menos escritura, es menos distraída y no puede provocar colisiones *de símbolos.* Las colisiones de símbolos son errores causados cuando distintos archivos de encabezado usan el mismo símbolo de preprocesador que el valor de protección. No forma parte del estándar de C++, pero lo implementan varios compiladores comunes.

No hay ninguna ventaja de usar la expresión include guard y `#pragma once` en el mismo archivo. El compilador reconoce la expresión de protección de include e implementa la optimización de varios elementos de la misma manera que la directiva si no hay ninguna directiva de preprocesador o código que no sea de comentario antes o después de la forma estándar de la `#pragma once` expresión:

```
// header.h
// Demonstration of the #include guard idiom.
// Note that the defined symbol can be arbitrary.
#ifndef HEADER_H_     // equivalently, #if !defined HEADER_H_
#define HEADER_H_
// Code placed here is included only once per translation unit
#endif // HEADER_H_
```

Se recomienda incluir la expresión de protección cuando el código debe ser portátil a los compiladores que no implementan la directiva, para mantener la coherencia con el código existente o cuando la optimización de varios incluir `#pragma once` es imposible. Puede producirse en proyectos complejos cuando el alias del sistema de archivos o las rutas de acceso de include con alias impiden que el compilador identifique archivos de incluir idénticos por ruta de acceso canónica.

Tenga cuidado de no usar ni incluir expresiones de protección en archivos de encabezado diseñados para incluirse varias veces, que usan símbolos `#pragma once` de preprocesador para controlar sus efectos. Para obtener un ejemplo de este diseño, vea el <assert.h> archivo de encabezado. Tenga también cuidado de administrar las rutas de acceso de include para evitar la creación de varias rutas de acceso a los archivos incluidos, lo que puede dar lugar a la optimización de varios elementos de incluyeción para las instancias de Include Guard y `#pragma once` .

Vea también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

`optimize` pragma

16/08/2021 • 2 minutes to read

Especifica optimizaciones en función de función por función.

Sintaxis

```
#pragma optimize( " [ optimization-list ] ", { on | off } )
```

Comentarios

debe `optimize` pragma aparecer fuera de una función. Tiene efecto en la primera función definida después de pragma que se ve . Los argumentos y activan o desactivan las opciones especificadas en la lista `on` `off` de *optimización.*

La *lista de optimización* puede ser cero o más de los parámetros que se muestran en la tabla siguiente.

Parámetros de la directiva pragma optimize

PARÁMETROS	TIPO DE OPTIMIZACIÓN
`g`	Habilitar optimizaciones globales.
`s` o `t`	Especificar secuencias cortas o rápidas de código máquina.
`y`	Generar punteros de marco en la pila del programa.

Estos parámetros son las mismas letras que se usan con las `/O` opciones del compilador. Por ejemplo, lo siguiente pragma es equivalente a la opción del `/Os` compilador:

```
#pragma optimize( "s", on )
```

El uso `optimize` pragma de con la cadena vacía ( ) `""` es una forma especial de la directiva :

Cuando se usa el parámetro , se desactivan todas `off` las `g` `s` `t` optimizaciones, , , y `y` .

Cuando se usa el parámetro , restablece las optimizaciones a las que `on` especificó mediante la opción `/O` del compilador.

```
#pragma optimize( "", off )
/* unoptimized code section */
#pragma optimize( "", on )
```

Vea también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

pack pragma

17/08/2021 • 4 minutes to read

Especifica la alineación de empaquetado para los miembros de estructura, unión y clase.

Sintaxis

```
#pragma pack( show )
#pragma pack( push [ , identifier ] [ , n ] )
#pragma pack( pop [ , { identifier | n } ] )
#pragma pack( [ n ] )
```

Parámetros

`show`
(Opcional) Muestra el valor de byte actual para la alineación de empaquetado. El valor se muestra mediante un mensaje de advertencia.

`push`
(Opcional) Inserta el valor de alineación de empaquetado actual en la pila interna del compilador y establece el valor de alineación de empaquetado actual en *n*. Si no se especifica *n,* se inserta el valor de alineación de empaquetado actual.

`pop`
(Opcional) Quita el registro de la parte superior de la pila interna del compilador. Si no se especifica *n* con , el valor de empaquetado asociado al registro resultante en la parte superior de la pila es el nuevo valor `pop` de alineación de empaquetado. Si se especifica *n,* por ejemplo, `#pragma pack(pop, 16)` , n *se* convierte en el nuevo valor de alineación de empaquetado. Si usa un elemento , por ejemplo, , todos los registros de la pila se abren hasta que se encuentra `identifier` el registro que se ha `#pragma pack(pop, r1)` `identifier` encontrado. Ese registro se saca y el valor de empaquetado asociado al registro que se encuentra en la parte superior de la pila se convierte en el nuevo valor de alineación de empaquetado. Si usa un elemento que no se encuentra en ningún registro de la `identifier` pila, se `pop` omite .

La instrucción `#pragma pack (pop, r1, 2)` es equivalente a seguido de `#pragma pack (pop, r1)` `#pragma pack(2)` .

`identifier`
(Opcional) Cuando se usa `push` con , asigna un nombre al registro en la pila interna del compilador. Cuando se usa con `pop` , quita los registros de la pila interna hasta que se `identifier` quita. Si `identifier` no se encuentra en la pila interna, no se hace nada.

`n`
(Opcional) Especifica el valor, en bytes, que se usará para el empaquetado. Si no se `/Zp` establece la opción del compilador para el módulo, el valor predeterminado de es `n` 8. Los valores válidos son 1, 2, 4, 8 y 16. La alineación de un miembro se encuentra en un límite que es un múltiplo de o un múltiplo del tamaño del `n` miembro, lo que sea menor.

Comentarios

Empaquetar una clase es colocar sus miembros directamente detrás de otros en la memoria. Esto puede

significar que algunos o todos los miembros se pueden alinear en un límite menor que la alineación predeterminada de la arquitectura de destino. `pack` proporciona control en el nivel de declaración de datos. Difiere de la opción del compilador `/Zp` , que solo proporciona control de nivel de módulo. **pack** tiene efecto en la primera `struct` declaración , o después de que se ve `union` `class` pragma . `pack` no tiene ningún efecto en las definiciones. Llamar `pack` a sin argumentos establece en el valor *`n`* establecido en la opción del compilador `/Zp` . Si no se establece la opción del compilador, el valor predeterminado es 8 para x86, ARM y ARM64. El valor predeterminado es 16 para x64 nativo.

Si cambia la alineación de una estructura, es posible que no use tanto espacio en la memoria. Sin embargo, es posible que vea una pérdida de rendimiento o incluso una excepción generada por hardware para el acceso no alineado. Puede modificar este comportamiento de excepción mediante `SetErrorMode` .

Para obtener más información sobre cómo modificar la alineación, consulte estos artículos:

- `alignof`
- `align`
- `__unaligned`
- Ejemplos de alineación de estructura (específico de x64)

> **WARNING**
> En Visual Studio 2015 y versiones posteriores puede usar los operadores estándar y , que a diferencia de `alignas` `alignof` y son `__alignof` `__declspec( align )` portátiles entre compiladores. El estándar de C++ no aborda el empaquetado, por lo que debe seguir utilizando (o la extensión correspondiente en otros compiladores) para especificar alineaciones menores que el tamaño de palabra de la arquitectura `pack` de destino.

Ejemplos

En el ejemplo siguiente se muestra cómo usar `pack` pragma para cambiar la alineación de una estructura .

```
// pragma_directives_pack.cpp
#include <stddef.h>
#include <stdio.h>

struct S {
   int i;   // size 4
   short j;   // size 2
   double k;   // size 8
};

#pragma pack(2)
struct T {
   int i;
   short j;
   double k;
};

int main() {
   printf("%zu ", offsetof(S, i));
   printf("%zu ", offsetof(S, j));
   printf("%zu\n", offsetof(S, k));

   printf("%zu ", offsetof(T, i));
   printf("%zu ", offsetof(T, j));
   printf("%zu\n", offsetof(T, k));
}
```

```
0 4 8
0 4 6
```

En el ejemplo siguiente se muestra cómo usar la sintaxis *push*, *pop y show.*

```
// pragma_directives_pack_2.cpp
// compile with: /W1 /c
#pragma pack()   // n defaults to 8; equivalent to /Zp8
#pragma pack(show)   // C4810
#pragma pack(4)   // n = 4
#pragma pack(show)   // C4810
#pragma pack(push, r1, 16)   // n = 16, pushed to stack
#pragma pack(show)   // C4810

// pop to the identifier and then set
// the value of the current packing alignment:
#pragma pack(pop, r1, 2)   // n = 2 , stack popped
#pragma pack(show)   // C4810
```

Vea también

Directivas Pragma y las palabras `__pragma` `_Pragma` clave y

`pointers_to_members` pragma

11/08/2021 • 2 minutes to read

Específicos de C++

Especifica si un puntero a un miembro de clase se puede declarar antes de su definición de clase asociada. Se usa para controlar el tamaño del puntero y el código necesario para interpretar el puntero.

Sintaxis

```
#pragma pointers_to_members( best_case )
#pragma pointers_to_members( full_generality [ , most-general-representation ] )
```

Comentarios

Puede colocar un en el archivo de código fuente como alternativa al uso de las opciones del compilador `pointers_to_members` pragma `/vmb` o `/vmg` y `/vmm` , `/vms` o `/vmv` las palabras clave de herencia específicas deMicrosoft .

El argumento de declaración de puntero especifica si ha declarado un puntero a un miembro antes o después de la definición de función asociada. El *`pointer-declaration`* argumento es uno de estos dos símbolos:

- `full_generality`
  Genera código seguro, a veces no optimo. Use `full_generality` si cualquier puntero a un miembro se declara antes de la definición de clase asociada. Este argumento siempre usa la representación de puntero especificada por el *`most-general-representation`* argumento . Equivalente a `/vmg` .

- `best_case`
  Genera código óptimo mediante la mejor representación de mayúsculas y minúsculas para todos los punteros a miembros. Requiere que defina la clase antes de declarar un puntero a un miembro. El valor predeterminado es `best_case` .

El argumento especifica la representación de puntero más pequeña que el compilador debe usar para hacer referencia de forma segura a cualquier puntero a un miembro de una *`most-general-representation`* clase en una unidad de traducción. El argumento puede ser uno de estos valores:

- `single_inheritance`
  La representación más general es el puntero de herencia única a la función miembro. Equivalente a `/vmg /vms` . Produce un error si el modelo de herencia de una definición de clase es múltiple o virtual.

- `multiple_inheritance`
  La representación más general es el puntero de herencia múltiple a la función miembro. Equivalente a `/vmg /vmm` . Produce un error si el modelo de herencia de una definición de clase es virtual.

- `virtual_inheritance`
  La representación más general es el puntero de herencia virtual a la función miembro. Equivalente a `/vmg /vmv` . Nunca produce un error. `virtual_inheritance` es el argumento predeterminado cuando `#pragma pointers_to_members(full_generality)` se usa .

Caution

Se recomienda colocar el único en el archivo de código fuente al que desea afectar y solo después

`pointers_to_members` pragma de cualquier `#include` directiva. Esta práctica reduce el riesgo de que afecte a otros archivos y que especifique accidentalmente varias definiciones para la misma variable, función o nombre pragma de clase.

Ejemplo

```
//   Specify single-inheritance only
#pragma pointers_to_members( full_generality, single_inheritance )
```

Específicos de C++: END

Consulte también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

`pop_macro` pragma

12/08/2021 • 2 minutes to read

Establece el valor de la *macro de nombre* de macro en el valor de la parte superior de la pila para esta macro.

Sintaxis

```
#pragma pop_macro( "macro-name" )
```

Comentarios

Se debe emitir un para `push_macro` *nombre* de macro antes de poder realizar una `pop_macro` .

Ejemplo

```
// pragma_directives_pop_macro.cpp
// compile with: /W1
#include <stdio.h>
#define X 1
#define Y 2

int main() {
   printf("%d",X);
   printf("\n%d",Y);
   #define Y 3   // C4005
   #pragma push_macro("Y")
   #pragma push_macro("X")
   printf("\n%d",X);
   #define X 2   // C4005
   printf("\n%d",X);
   #pragma pop_macro("X")
   printf("\n%d",X);
   #pragma pop_macro("Y")
   printf("\n%d",Y);
}
```

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

Consulte también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

`push_macro` pragma

17/08/2021 • 2 minutes to read

Guarda el valor de la *macro de nombre* de macro en la parte superior de la pila para esta macro.

Sintaxis

```
#pragma push_macro("macro-name")
```

Comentarios

Puede recuperar el valor del *nombre de macro* con `pop_macro` .

`pop_macro` Consulte pragma para obtener un ejemplo.

Vea también

Directivas Pragma y las palabras `__pragma` `_Pragma` clave y

`region` y `endregion` pragma

12/08/2021 • 2 minutes to read

`#pragma region` permite especificar un bloque de código que se puede expandir o contraer al usar la característica de línea del editor Visual Studio texto.

Sintaxis

```
** #pragma region **name
** #pragma endregion **comentario
```

Parámetros

Comentario
(Opcional) Comentario que se mostrará en el editor de código.

Nombre
(Opcional) Nombre de la región. Este nombre se muestra en el editor de código.

Comentarios

`#pragma endregion` marca el final de un `#pragma region` bloque.

Una `#pragma region` directiva debe terminar un `#pragma endregion` bloque.

Ejemplo

```
// pragma_directives_region.cpp
#pragma region Region_1
void Test() {}
void Test2() {}
void Test3() {}
#pragma endregion Region_1

int main() {}
```

Consulte también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

`runtime_checks` pragma

17/08/2021 • 2 minutes to read

Deshabilita o restaura la configuración de `/RTC` la opción del compilador.

Sintaxis

```
#pragma runtime_checks( " [ runtime-check-options ] ", { restore | off } )
```

Comentarios

No se puede habilitar una comprobación en tiempo de ejecución que no esté habilitada por una opción del compilador. Por ejemplo, si no especifica en la línea de `/RTCs` comandos, la especificación no habilitará la comprobación del `#pragma runtime_checks( "s", restore)` marco de pila.

debe `runtime_checks` pragma aparecer fuera de una función y tiene efecto en la primera función definida después pragma de que se ve . Los `restore` `off` argumentos y activan o desactivan las opciones `runtime_checks` pragma especificadas en .

Las *opciones de comprobación en tiempo de* ejecución pueden ser cero o más de los parámetros que se muestran en la tabla siguiente.

Parámetros de la directiva pragma runtime_checks

PARÁMETROS	TIPO DE COMPROBACIÓN EN TIEMPO DE EJECUCIÓN
`s`	Habilita la comprobación de pila (marco).
`c`	Comunica los casos en que se asigna un valor a un tipo de datos más pequeño y se provoca una pérdida de datos.
`u`	Notifica cuándo se usa una variable antes de definirla.

Estos parámetros son los mismos que se usan con la opción `/RTC` del compilador. Por ejemplo:

```
#pragma runtime_checks( "sc", restore )
```

El uso `runtime_checks` pragma de con la cadena vacía ( ) `""` es una forma especial de la directiva :

- Cuando se usa el parámetro , desactiva las comprobaciones de errores en tiempo de ejecución enumeradas `off` en la tabla anterior.
- Cuando se usa el parámetro , restablece las comprobaciones de errores en tiempo de ejecución a las que especificó `restore` mediante la opción del `/RTC` compilador.

```
#pragma runtime_checks( "", off )
/* runtime checks are off in this region */
#pragma runtime_checks( "", restore )
```

Vea también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

`section` pragma

16/08/2021 • 2 minutes to read

Crea una sección en un archivo OBJ.

Sintaxis

```
#pragma section( " section-name " [ , attributes ] )
```

Comentarios

Los *términos segment y section* tienen el mismo significado en este artículo.

Una vez definida una sección, sigue siendo válida para el resto de la compilación. Sin embargo, debe usar `__declspec(allocate)` o no se coloca nada en la sección .

section-name es un parámetro necesario que se convierte en el nombre de la sección. El nombre no debe estar en conflicto con ningún nombre de sección estándar. Consulte `/SECTION` para obtener una lista de nombres que no debe usar al crear una sección.

attributes es un parámetro opcional que consta de uno o varios atributos separados por comas que se asignarán a la sección. Los *atributos posibles* son:

ATRIBUTO	DESCRIPCIÓN
`read`	Permite operaciones de lectura en los datos.
`write`	Permite operaciones de escritura en los datos.
`execute`	Permite que el código se ejecute.
`shared`	Comparte la sección entre todos los procesos que cargan la imagen.
`nopage`	Marca la sección como no paginable. Útil para controladores de dispositivos Win32.
`nocache`	Marca la sección como no almacenable en caché. Útil para controladores de dispositivos Win32.
`discard`	Marca la sección como descartable. Útil para controladores de dispositivos Win32.
`remove`	Marca la sección como no residentes en memoria. Solo para controladores de dispositivos virtuales (V *x* D).

Si no especifica ningún atributo, la sección tiene los `read` atributos `write` y .

Ejemplo

En este ejemplo, la primera sección pragma identifica la sección y sus atributos. El entero `j` no se coloca en

porque no se `mysec` declaró mediante `__declspec(allocate)` . En su `j` lugar, va a la sección de datos. El entero `i` entra en debido a su atributo de clase de `mysec` `__declspec(allocate)` almacenamiento.

```
// pragma_section.cpp
#pragma section("mysec",read,write)
int j = 0;

__declspec(allocate("mysec"))
int i = 0;

int main(){}
```

Vea también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

`setlocale` pragma

16/08/2021 • 2 minutes to read

Define la *configuración regional,* el país, la región y el idioma que se usarán al traducir constantes de caracteres anchos y literales de cadena.

Sintaxis

```
#pragma setlocale( " [ locale-string ] " )
```

Comentarios

Dado que el algoritmo para convertir caracteres multibyte en caracteres anchos puede variar según la configuración regional, o la compilación puede tener lugar en una configuración regional diferente de donde se ejecutará un archivo ejecutable, esto proporciona una manera de especificar la configuración regional de destino en tiempo de pragma compilación. Garantiza que las cadenas de caracteres anchos se almacenan en el formato correcto.

La cadena *de configuración regional predeterminada* es la cadena vacía, especificada por `#pragma setlocale( "" )` .

La `"C"` configuración regional asigna cada carácter de la cadena a su valor como `wchar_t` . Otros valores válidos `setlocale` para son las entradas que se encuentran en la lista Cadenas de lenguaje. Por ejemplo, podría especificar:

```
#pragma setlocale("dutch")
```

La capacidad de especificar una cadena de idioma depende de la página de códigos y la compatibilidad con el identificador de idioma en el equipo.

Vea también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

`strict_gs_check` pragma

14/08/2021 • 2 minutes to read

Esto pragma proporciona una comprobación de seguridad mejorada.

Sintaxis

```
#pragma strict_gs_check( [ push, ] { on | off } )
#pragma strict_gs_check( pop )
```

Comentarios

Indica al compilador que inserte una cookie aleatoria en la pila de la función para ayudar a detectar algunas categorías de saturación del búfer basada en la pila. De forma predeterminada, `/GS` la opción del compilador no inserta una cookie para todas las funciones. Para obtener más información, vea `/GS` (Comprobación de seguridad del búfer).

Compile mediante `/GS` para habilitar `strict_gs_check` .

Úsese pragma esto en módulos de código expuestos a datos potencialmente peligrosos. `strict_gs_check` es un agresivo y se aplica a funciones que podrían no necesitar esta defensa, pero está optimizada para minimizar su efecto en el rendimiento de pragma la aplicación resultante.

Incluso si usa este pragma , debe procurar escribir código seguro. Es decir, asegúrese de que el código no tiene saturaciones de búfer. `strict_gs_check` puede proteger la aplicación frente a saturaciones de búfer que permanecen en el código.

Ejemplo

En este ejemplo, se produce una saturación del búfer cuando se copia una matriz en una matriz local. Al compilar este código con , no se inserta ninguna cookie en la pila, porque el tipo de datos de la `/GS` matriz es un puntero. Al agregar `strict_gs_check` pragma , se fuerza la cookie de pila en la pila de funciones.

```
// pragma_strict_gs_check.cpp
// compile with: /c

#pragma strict_gs_check(on)

void ** ReverseArray(void **pData,
                     size_t cData)
{
    // *** This buffer is subject to being overrun!! ***
    void *pReversed[20];

    // Reverse the array into a temporary buffer
    for (size_t j = 0, i = cData; i ; --i, ++j)
        // *** Possible buffer overrun!! ***
            pReversed[j] = pData[i];

    // Copy temporary buffer back into input/output buffer
    for (size_t i = 0; i < cData ; ++i)
        pData[i] = pReversed[i];

    return pData;
}
```

Vea también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y `/GS` (Comprobación de seguridad del búfer)

`system_header` pragma

14/08/2021 • 2 minutes to read

Trate el resto del archivo como externo para los informes de diagnóstico.

Sintaxis

```
#pragma system_header
```

Comentarios

indica `system_header` pragma al compilador que muestre los diagnósticos en el nivel especificado por `/external:Wn` la opción para el resto del archivo de origen actual. Para obtener más información sobre cómo especificar archivos externos y el nivel de advertencia externo para el compilador, vea `/external` .

no `system_header` pragma se aplica más allá del final del archivo de código fuente actual. En otras palabras, no se aplica a los archivos que incluyen este archivo. se aplica incluso si no se especifica ningún `system_header` pragma otro archivo como externo al compilador. Sin embargo, si no se especifica ningún nivel de opción, el compilador puede emitir un diagnóstico y usa el mismo nivel de advertencia que se aplica a archivos `/external:Wn` no externos. Otras pragma directivas que afectan al comportamiento de advertencia se siguen aplicando después de `system_header` pragma . El efecto de `#pragma system_header` es similar a `warning pragma` :

```
// If n represents the warning level specified by /external:Wn,
// #pragma system_header is roughly equivalent to:
#pragma warning( push, n )

// . . .

// At the end of the file:
#pragma warning( pop )
```

está `system_header` pragma disponible a partir de Visual Studio versión 16.10 de 2019.

Ejemplo

Este encabezado de ejemplo muestra cómo marcar el contenido de un archivo como externo:

```
// library.h
// Use /external:Wn to set the compiler diagnostics level for this file's contents

#pragma once
#ifndef _LIBRARY_H // include guard for 3rd party interop
#define _LIBRARY_H
#pragma system_header
// The compiler applies the /external:Wn diagnostic level from here to the end of this file.

// . . .

// You can still override the external diagnostic level for warnings locally:
#pragma warning( push )
#pragma warning( error : 4164 )

// . . .

#pragma warning(pop)

// . . .

#endif
```

Vea también

`/external`
`warning pragma`
`/Wn` (Nivel de advertencia del compilador)
Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

`vtordisp` pragma

12/08/2021 • 2 minutes to read

Controla la adición del miembro de `vtordisp` desplazamiento de construcción o destrucción oculto. es `vtordisp` pragma específico de C++.

Sintaxis

```
#pragma vtordisp( [ push, ] n** ) **
#pragma vtordisp(pop)
#pragma vtordisp()
#pragma vtordisp( [ push, ] { on | off } )
```

Parámetros

`push`
Inserta la configuración `vtordisp` actual en la pila interna del compilador y establece la nueva configuración en `vtordisp` *n*. Si *no* se especifica n, la configuración `vtordisp` actual no se modifica.

`pop`
Quita el registro superior de la pila interna del compilador y restaura la `vtordisp` configuración al valor quitado.

N
Especifica el nuevo valor para la `vtordisp` configuración. Los valores posibles `0` son , o , `1` `2` correspondientes a las opciones del `/vd0` compilador , y `/vd1` `/vd2` . Para obtener más información, vea `/vd` (Deshabilitar desplazamientos de construcción).

`on`
Equivalente a `#pragma vtordisp(1)` .

`off`
Equivalente a `#pragma vtordisp(0)` .

Comentarios

solo `vtordisp` pragma es aplicable al código que usa bases virtuales. Si una clase derivada invalida una función virtual que hereda de una clase base virtual, y si un constructor o destructor de la clase derivada llama a esa función mediante un puntero a la clase base virtual, el compilador podría introducir campos ocultos adicionales en las clases con `vtordisp` bases virtuales.

afecta `vtordisp` pragma al diseño de las clases que lo siguen. Las `/vd0` opciones del compilador , y especifican el mismo comportamiento para los `/vd1` `/vd2` módulos completos. Especifica o `0` `off` suprime los miembros `vtordisp` ocultos. Desactive solo si no hay ninguna posibilidad de que los constructores y destructores de la clase llamen a funciones virtuales en el objeto al que `vtordisp` apunta el `this` puntero.

Al especificar `1` o , el valor `on` predeterminado, se habilitan los `vtordisp` miembros ocultos cuando son necesarios.

La `2` especificación habilita los miembros `vtordisp` ocultos para todas las bases virtuales con funciones virtuales. `#pragma vtordisp(2)` puede ser necesario para garantizar el rendimiento correcto de `dynamic_cast` en

un objeto construido parcialmente. Para obtener más información, vea Advertencia del compilador (nivel 1) C4436.

`#pragma vtordisp()`, sin argumentos, restaura la `vtordisp` configuración a su configuración inicial.

```
#pragma vtordisp(push, 2)
class GetReal : virtual public VBase { ... };
#pragma vtordisp(pop)
```

Consulte también

Directivas Pragma y las `__pragma` palabras clave `_Pragma` y

`warning` pragma

13/08/2021 • 3 minutes to read

Habilita la modificación selectiva del comportamiento de los mensajes de advertencia del compilador.

Sintaxis

```
#pragma warning(
    warning-specifier : warning-number-list
    [ ; warning-specifier : warning-number-list ... ] )
#pragma warning( push [ , n ] )
#pragma warning( pop )
```

Comentarios

Los siguientes parámetros de warning-specifier están disponibles.

WARNING-SPECIFIER	SIGNIFICADO
`1`, `2`, `3`, `4`	Aplique el nivel especificado a las advertencias especificadas. También activa una advertencia especificada que está desactivada de forma predeterminada.
`default`	Restablece el comportamiento de advertencia a su valor predeterminado. También activa una advertencia especificada que está desactivada de forma predeterminada. La advertencia se generará en su nivel predeterminado documentado. Para obtener más información, vea Advertencias del compilador desactivadas de forma predeterminada.
`disable`	No emita los mensajes de advertencia especificados.
`error`	Notifica las advertencias especificadas como errores.
`once`	Muestra los mensajes especificados solo una vez.
`suppress`	Inserta el estado actual de en la pila, deshabilita la advertencia especificada para la línea siguiente y, a continuación, muestra la pila de advertencias para que se pragma pragma restablezca el estado.

La siguiente instrucción de código muestra que un parámetro puede contener varios números de advertencia y que se pueden especificar varios parámetros *warning-number-list* *warning-specifier* en la misma pragma directiva.

```
#pragma warning( disable : 4507 34; once : 4385; error : 164 )
```

Esta directiva es funcionalmente equivalente al código siguiente:

```
// Disable warning messages 4507 and 4034.
#pragma warning( disable : 4507 34 )

// Issue warning C4385 only once.
#pragma warning( once : 4385 )

// Report warning C4164 as an error.
#pragma warning( error : 164 )
```

El compilador agrega 4000 a cualquier número de advertencia que esté entre 0 y 999.

Los números de advertencia del intervalo 4700-4999 están asociados a la generación de código. Para estas advertencias, el estado de la advertencia en vigor cuando el compilador alcanza la definición de función permanece en vigor para el resto de la función. El uso de en la función para cambiar el estado de un número de advertencia mayor que `warning` 4699 solo tiene efecto después del pragma final de la función. En el ejemplo siguiente se muestra la ubicación correcta de para deshabilitar un mensaje de advertencia de generación de código `warning` pragma y, a continuación, para restaurarlo.

```
// pragma_warning.cpp
// compile with: /W1
#pragma warning(disable:4700)
void Test() {
   int x;
   int y = x;   // no C4700 here
   #pragma warning(default:4700)   // C4700 enabled after Test ends
}

int main() {
   int x;
   int y = x;   // C4700
}
```

Observe que, en todo el cuerpo de una función, la última configuración de `warning` pragma estará en vigor para toda la función.

Inserción y extracción

también `warning` pragma admite la sintaxis siguiente, donde el *parámetro n* opcional representa un nivel de advertencia (del 1 al 4).

`#pragma warning( push [ , n ] )`

`#pragma warning( pop )`

almacena pragma `warning( push )` el estado de advertencia actual para cada advertencia. almacena pragma `warning( push, n )` el estado actual de cada advertencia y establece el nivel de advertencia global en *n*.

muestra pragma `warning( pop )` el último estado de advertencia que se inserta en la pila. Los cambios realizados en el estado de advertencia entre `push` y `pop` se desconvierte. Considere este ejemplo:

```
#pragma warning( push )
#pragma warning( disable : 4705 )
#pragma warning( disable : 4706 )
#pragma warning( disable : 4707 )
// Some code
#pragma warning( pop )
```

Al final de este código, restaura el estado de cada advertencia `pop` (incluye 4705, 4706 y 4707) a lo que era al

principio del código.

Al escribir archivos de encabezado, puede usar y para garantizar que los cambios de estado de advertencia realizados por un usuario no impidan que los encabezados se `push` `pop` compilan correctamente. Use `push` al principio del encabezado y al `pop` final. Por ejemplo, puede tener un encabezado que no se compila correctamente en el nivel de advertencia 4. El código siguiente cambia el nivel de advertencia a 3 y, a continuación, restaura el nivel de advertencia original al final del encabezado.

```
#pragma warning( push, 3 )
// Declarations/definitions
#pragma warning( pop )
```

Para obtener más información sobre las opciones del compilador que le ayudan a suprimir advertencias, vea `/FI` y `/w` .

Vea también

Directivas Pragma y las palabras `__pragma` `_Pragma` clave y