C/C++ preprocessor reference

Article • 08/03/2021

The *C/C++ preprocessor reference* explains the preprocessor as it is implemented in Microsoft C/C++. The preprocessor performs preliminary operations on C and C++ files before they are passed to the compiler. You can use the preprocessor to conditionally compile code, insert files, specify compile-time error messages, and apply machine-specific rules to sections of code.

In Visual Studio 2019 the /Zc:preprocessor compiler option provides a fully conformant C11 and C17 preprocessor. This is the default when you use the compiler flag `/std:c11` or `/std:c17`.

In this section

Preprocessor
Provides an overview of the traditional and new conforming preprocessors.

Preprocessor directives
Describes directives, typically used to make source programs easy to change and easy to compile in different execution environments.

Preprocessor operators
Discusses the four preprocessor-specific operators used in the context of the `#define` directive.

Predefined macros
Discusses predefined macros as specified by the C and C++ standards and by Microsoft C++.

Pragmas
Discusses pragmas, which offer a way for each compiler to offer machine- and operating system-specific features while retaining overall compatibility with the C and C++ languages.

Related sections

C++ language reference
Provides reference material for the Microsoft implementation of the C++ language.

C language reference

Provides reference material for the Microsoft implementation of the C language.

C/C++ build reference

Provides links to topics discussing compiler and linker options.

Visual Studio projects - C++

Describes the user interface in Visual Studio that enables you to specify the directories that the project system will search to locate files for your C++ project.

Preprocessor

Article • 08/03/2021

The preprocessor is a text processor that manipulates the text of a source file as part of the first phase of translation. The preprocessor doesn't parse the source text, but it does break it up into tokens to locate macro calls. Although the compiler ordinarily invokes the preprocessor in its first pass, the preprocessor can also be invoked separately to process text without compiling.

The reference material on the preprocessor includes the following sections:

- Preprocessor directives
- Preprocessor operators
- Predefined macros
- Pragmas

Microsoft Specific

You can obtain a listing of your source code after preprocessing by using the /E or /EP compiler option. Both options invoke the preprocessor and send the resulting text to the standard output device, which, in most cases, is the console. The difference between the two options is that `/E` includes `#line` directives, and `/EP` strips out these directives.

END Microsoft Specific

Special terminology

In the preprocessor documentation, the term "argument" refers to the entity that is passed to a function. In some cases, it's modified by "actual" or "formal," which describes the argument expression specified in the function call, and the argument declaration specified in the function definition, respectively.

The term "variable" refers to a simple C-type data object. The term "object" refers to both C++ objects and variables; it's an inclusive term.

See also

C/C++ preprocessor reference
Phases of translation

MSVC new preprocessor overview

Article • 02/01/2022

We're updating the Microsoft C++ preprocessor to improve standards conformance, fix longstanding bugs, and change some behaviors that are officially undefined. We've also added new diagnostics to warn on errors in macro definitions.

Starting in Visual Studio 2019 version 16.5, preprocessor support for the C++20 standard is feature-complete. These changes are available by using the /Zc:preprocessor compiler switch. An experimental version of the new preprocessor is available in earlier versions starting in Visual Studio 2017 version 15.8. You can enable it by using the /experimental:preprocessor compiler switch. The default preprocessor behavior remains the same as in previous versions.

New predefined macro

You can detect which preprocessor is in use at compile time. Check the value of the predefined macro _MSVC_TRADITIONAL to tell if the traditional preprocessor is in use. This macro is set unconditionally by versions of the compiler that support it, independent of which preprocessor is invoked. Its value is 1 for the traditional preprocessor. It's 0 for the conforming preprocessor.

C++

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

Behavior changes in the new preprocessor

The initial work on the new preprocessor has been focused on making all macro expansions conform to the standard. It lets you use the MSVC compiler with libraries that are currently blocked by the traditional behaviors. We tested the updated preprocessor on real world projects. Here are some of the more common breaking changes we found:

Macro comments

The traditional preprocessor is based on character buffers rather than preprocessor tokens. It allows unusual behavior such as the following preprocessor comment trick, which doesn't work under the conforming preprocessor:

C++

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

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

The standards-conforming fix is to declare `int myVal` inside the appropriate `#ifdef/#endif` directives:

C++

```
#define MYVAL 1

#ifdef MYVAL
int myVal;
#endif
```

L#val

The traditional preprocessor incorrectly combines a string prefix to the result of the stringizing operator (#) operator:

C++

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

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

In this case, the `L` prefix is unnecessary because the adjacent string literals are combined after macro expansion anyway. The backward-compatible fix is to change the definition:

C++

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

The same issue is also found in convenience macros that "stringize" the argument to a wide string literal:

```
C++

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

You can fix the issue in various ways:

- Use string concatenation of `L""` and `#str` to add prefix. Adjacent string literals are combined after macro expansion:

  ```
  C++

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

- Add the prefix after `#str` is stringized with additional macro expansion

  ```
  C++

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

- Use the concatenation operator `##` to combine the tokens. The order of operations for `##` and `#` is unspecified, although all compilers seem to evaluate the `#` operator before `##` in this case.

  ```
  C++

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

Warning on invalid

When the token-pasting operator (##) doesn't result in a single valid preprocessing token, the behavior is undefined. The traditional preprocessor silently fails to combine the tokens. The new preprocessor matches the behavior of most other compilers and emits a diagnostic.

C++

```
// 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;
```

Comma elision in variadic macros

The traditional MSVC preprocessor always removes commas before empty `__VA_ARGS__` replacements. The new preprocessor more closely follows the behavior of other popular cross-platform compilers. For the comma to be removed, the variadic argument must be missing (not just empty) and it must be marked with a `##` operator. Consider the following example:

C++

```
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, );
}
```

In the following example, in the call to `FUNC2(1)` the variadic argument is missing in the macro being invoked. In the call to `FUNC2(1, )` the variadic argument is empty, but not missing (notice the comma in the argument list).

C++

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

   // Expands to func(1, )
```

```
    FUNC2(1, );
}
```

In the upcoming C++20 standard, this issue has been addressed by adding `__VA_OPT__`. New preprocessor support for `__VA_OPT__` is available starting in Visual Studio 2019 version 16.5.

C++20 variadic macro extension

The new preprocessor supports C++20 variadic macro argument elision:

C++

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

This code isn't conforming before the C++20 standard. In MSVC, the new preprocessor extends this C++20 behavior to lower language standard modes (`/std:c++14`, `/std:c++17`). This extension matches the behavior of other major cross-platform C++ compilers.

Macro arguments are "unpacked"

In the traditional preprocessor, if a macro forwards one of its arguments to another dependent macro then the argument doesn't get "unpacked" when it's inserted. Usually this optimization goes unnoticed, but it can lead to unusual behavior:

C++

```
// 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", };
```

When expanding `A()`, the traditional preprocessor forwards all of the arguments packaged in `__VA_ARGS__` to the first argument of TWO_STRINGS, which leaves the variadic argument of `TWO_STRINGS` empty. That causes the result of `#first` to be "1, 2" rather than just "1". If you're following along closely, then you may be wondering what happened to the result of `#__VA_ARGS__` in the traditional preprocessor expansion: if the variadic parameter is empty it should result in an empty string literal `""`. A separate issue kept the empty string literal token from being generated.

Rescanning replacement list for macros

After a macro is replaced, the resulting tokens are rescanned for additional macro identifiers to replace. The algorithm used by the traditional preprocessor for doing the rescan isn't conforming, as shown in this example based on actual code:

C++

```
#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");
```

Although this example may seem a bit contrived, we've seen it in real-world code.

To see what's going on, we can break down the expansion starting with `DO_THING`:

1. `DO_THING(1, "World")` expands to `CAT(IMPL, 1) ECHO(("Hello", "World"))`
2. `CAT(IMPL, 1)` expands to `IMPL ## 1`, which expands to `IMPL1`
3. Now the tokens are in this state: `IMPL1 ECHO(("Hello", "World"))`
4. The preprocessor finds the function-like macro identifier `IMPL1`. Since it's not followed by a `(`, it isn't considered a function-like macro invocation.
5. The preprocessor moves on to the following tokens. It finds the function-like macro `ECHO` gets invoked: `ECHO(("Hello", "World"))`, which expands to `("Hello",`

`"World")`

6. `IMPL1` is never considered again for expansion, so the full result of the expansions is: `IMPL1("Hello", "World");`

To modify the macro to behave the same way under both the new preprocessor and the traditional preprocessor, add another layer of indirection:

C++

```
#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");
```

Incomplete features before 16.5

Starting in Visual Studio 2019 version 16.5, the new preprocessor is feature-complete for C++20. In previous versions of Visual Studio, the new preprocessor is mostly complete, although some preprocessor directive logic still falls back to the traditional behavior. Here's a partial list of incomplete features in Visual Studio versions before 16.5:

- Support for `_Pragma`
- C++20 features
- Boost blocking bug: Logical operators in preprocessor constant expressions aren't fully implemented in the new preprocessor before version 16.5. On some `#if` directives, the new preprocessor can fall back to the traditional preprocessor. The effect is only noticeable when macros incompatible with the traditional preprocessor get expanded. It can happen when building Boost preprocessor slots.

Phases of translation

Article • 08/03/2021

C and C++ programs consist of one or more source files, each of which contains some of the text of the program. A source file, together with its *include files*, files that are included using the `#include` preprocessor directive, but not including sections of code removed by conditional-compilation directives such as `#if`, is called a *translation unit*.

Source files can be translated at different times. In fact, it's common to translate only out-of-date files. The translated translation units can be processed into separate object files or object-code libraries. These separate, translated translation units are then linked to form an executable program or a dynamic-link library (DLL). For more information about files that can be used as input to the linker, see LINK input files.

Translation units can communicate using:

- Calls to functions that have external linkage.
- Calls to class member functions that have external linkage.
- Direct modification of objects that have external linkage.
- Direct modification of files.
- Interprocess communication (for Microsoft Windows-based applications only).

The following list describes the phases in which the compiler translates files:

Character mapping
Characters in the source file are mapped to the internal source representation. Trigraph sequences are converted to single-character internal representation in this phase.

Line splicing
All lines ending in a backslash (\) immediately followed by a newline character are joined with the next line in the source file, forming logical lines from the physical lines. Unless it's empty, a source file must end in a newline character that's not preceded by a backslash.

Tokenization
The source file is broken into preprocessing tokens and white-space characters. Comments in the source file are replaced with one space character each. Newline characters are retained.

Preprocessing
Preprocessing directives are executed and macros are expanded into the source file. The `#include` statement invokes translation starting with the preceding three translation steps on any included text.

Character-set mapping
All source character set members and escape sequences are converted to their equivalents in the execution character set. For Microsoft C and C++, both the source and the execution character sets are ASCII.

String concatenation
All adjacent string and wide-string literals are concatenated. For example, `"String "` `"concatenation"` becomes `"String concatenation"`.

Translation
All tokens are analyzed syntactically and semantically; these tokens are converted into object code.

Linkage
All external references are resolved to create an executable program or a dynamic-link library.

The compiler issues warnings or errors during phases of translation in which it encounters syntax errors.

The linker resolves all external references and creates an executable program or DLL by combining one or more separately processed translation units along with standard libraries.

See also

Preprocessor

Preprocessor directives

Article • 08/03/2021

Preprocessor directives, such as `#define` and `#ifdef`, are typically used to make source programs easy to change and easy to compile in different execution environments. Directives in the source file tell the preprocessor to take specific actions. For example, the preprocessor can replace tokens in the text, insert the contents of other files into the source file, or suppress compilation of part of the file by removing sections of text. Preprocessor lines are recognized and carried out before macro expansion. Therefore, if a macro expands into something that looks like a preprocessor command, it isn't recognized by the preprocessor.

Preprocessor statements use the same character set as source file statements, with the exception that escape sequences aren't supported. The character set used in preprocessor statements is the same as the execution character set. The preprocessor also recognizes negative character values.

The preprocessor recognizes the following directives:

#define
#elif
#else
#endif

#error
#if
#ifdef
#ifndef

#import
#include
#line

#pragma
#undef
#using

The number sign (`#`) must be the first nonwhite-space character on the line containing the directive. White-space characters can appear between the number sign and the first letter of the directive. Some directives include arguments or values. Any text that follows a directive (except an argument or value that is part of the directive) must be preceded

by the single-line comment delimiter (`//`) or enclosed in comment delimiters (`/* */`). Lines containing preprocessor directives can be continued by immediately preceding the end-of-line marker with a backslash (`\`).

Preprocessor directives can appear anywhere in a source file, but they apply only to the rest of the source file, after they appear.

See also

Preprocessor operators
Predefined macros
c/c++ preprocessor reference

#define directive (C/C++)

Article • 08/03/2021

The **#define** creates a *macro*, which is the association of an identifier or parameterized identifier with a token string. After the macro is defined, the compiler can substitute the token string for each occurrence of the identifier in the source file.

Syntax

> **#define** *identifier token-string*$_{opt}$
> **#define** *identifier* **(** *identifier*$_{opt}$**,** ... **,** *identifier*$_{opt}$ **)** *token-string*$_{opt}$

Remarks

The **#define** directive causes the compiler to substitute *token-string* for each occurrence of *identifier* in the source file. The *identifier* is replaced only when it forms a token. That is, *identifier* is not replaced if it appears in a comment, in a string, or as part of a longer identifier. For more information, see Tokens.

The *token-string* argument consists of a series of tokens, such as keywords, constants, or complete statements. One or more white-space characters must separate *token-string* from *identifier*. This white space is not considered part of the substituted text, nor is any white space that follows the last token of the text.

A `#define` without a *token-string* removes occurrences of *identifier* from the source file. The *identifier* remains defined and can be tested by using the `#if defined` and `#ifdef` directives.

The second syntax form defines a function-like macro with parameters. This form accepts an optional list of parameters that must appear in parentheses. After the macro is defined, each subsequent occurrence of *identifier*(*identifier*$_{opt}$, ..., *identifier*$_{opt}$) is replaced with a version of the *token-string* argument that has actual arguments substituted for formal parameters.

Formal parameter names appear in *token-string* to mark the locations where actual values are substituted. Each parameter name can appear multiple times in *token-string*, and the names can appear in any order. The number of arguments in the call must match the number of parameters in the macro definition. Liberal use of parentheses guarantees that complex actual arguments are interpreted correctly.

The formal parameters in the list are separated by commas. Each name in the list must be unique, and the list must be enclosed in parentheses. No spaces can separate *identifier* and the opening parenthesis. Use line concatenation — place a backslash (`\`) immediately before the newline character — for long directives on multiple source lines. The scope of a formal parameter name extends to the new line that ends *token-string*.

When a macro has been defined in the second syntax form, subsequent textual instances followed by an argument list indicate a macro call. The actual arguments that follows an instance of *identifier* in the source file are matched to the corresponding formal parameters in the macro definition. Each formal parameter in *token-string* that is not preceded by a stringizing (`#`), charizing (`#@`), or token-pasting (`##`) operator, or not followed by a `##` operator, is replaced by the corresponding actual argument. Any macros in the actual argument are expanded before the directive replaces the formal parameter. (The operators are described in Preprocessor operators.)

The following examples of macros with arguments illustrate the second form of the **#define** syntax:

```
C

// 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))
```

Arguments with side effects sometimes cause macros to produce unexpected results. A given formal parameter may appear more than one time in *token-string*. If that formal parameter is replaced by an expression with side effects, the expression, with its side effects, may be evaluated more than one time. (See the examples under Token-Pasting Operator (##).)

The `#undef` directive causes an identifier's preprocessor definition to be forgotten. See The #undef Directive for more information.

If the name of the macro being defined occurs in *token-string* (even as a result of another macro expansion), it is not expanded.

A second **#define** for a macro with the same name generates a warning unless the second token sequence is identical to the first.

Microsoft Specific

Microsoft C/C++ lets you redefine a macro if the new definition is syntactically identical to the original definition. In other words, the two definitions can have different parameter names. This behavior differs from ANSI C, which requires that the two definitions be lexically identical.

For example, the following two macros are identical except for the parameter names. ANSI C does not allow such a redefinition, but Microsoft C/C++ compiles it without error.

```
C

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

On the other hand, the following two macros are not identical and will generate a warning in Microsoft C/C++.

```
C

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

END Microsoft Specific

This example illustrates the **#define** directive:

```
C

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

The first statement defines the identifier `WIDTH` as the integer constant 80 and defines `LENGTH` in terms of `WIDTH` and the integer constant 10. Each occurrence of `LENGTH` is replaced by (`WIDTH + 10`). In turn, each occurrence of `WIDTH + 10` is replaced by the expression (`80 + 10`). The parentheses around `WIDTH + 10` are important because they control the interpretation in statements such as the following:

```
C

var = LENGTH * 20;
```

After the preprocessing stage the statement becomes:

```
C
```

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

which evaluates to 1800. Without parentheses, the result is:

```
C

var = 80 + 10 * 20;
```

which evaluates to 280.

Microsoft Specific

Defining macros and constants with the /D compiler option has the same effect as using a **#define** preprocessing directive at the start of your file. Up to 30 macros can be defined by using the /D option.

END Microsoft Specific

See also

Preprocessor directives

#error directive (C/C++)

Article • 08/03/2021

The **#error** directive emits a user-specified error message at compile time, and then terminates the compilation.

Syntax

> **#error** *token-string*

Remarks

The error message that this directive emits includes the *token-string* parameter. The *token-string* parameter is not subject to macro expansion. This directive is most useful during preprocessing, to notify the developer of a program inconsistency, or the violation of a constraint. The following example demonstrates error processing during preprocessing:

C++

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

See also

Preprocessor directives

#if, #elif, #else, and #endif directives (C/C++)

Article • 01/25/2023

The **#if** directive, with the **#elif**, **#else**, and **#endif** directives, controls compilation of portions of a source file. If the expression you write (after the **#if**) has a nonzero value, the line group immediately following the **#if** directive is kept in the translation unit.

Grammar

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

Remarks

Each **#if** directive in a source file must be matched by a closing **#endif** directive. Any number of **#elif** directives can appear between the **#if** and **#endif** directives, but at most

one **#else** directive is allowed. The **#else** directive, if present, must be the last directive before **#endif**.

The **#if**, **#elif**, **#else**, and **#endif** directives can nest in the *text* portions of other **#if** directives. Each nested **#else**, **#elif**, or **#endif** directive belongs to the closest preceding **#if** directive.

All conditional-compilation directives, such as **#if** and **#ifdef**, must match a closing **#endif** directive before the end of file. Otherwise, an error message is generated. When conditional-compilation directives are contained in include files, they must satisfy the same conditions: There must be no unmatched conditional-compilation directives at the end of the include file.

Macro replacement is done within the part of the line that follows an **#elif** command, so a macro call can be used in the *constant-expression*.

The preprocessor selects one of the given occurrences of *text* for further processing. A block specified in *text* can be any sequence of text. It can occupy more than one line. Usually *text* is program text that has meaning to the compiler or the preprocessor.

The preprocessor processes the selected *text* and passes it to the compiler. If *text* contains preprocessor directives, the preprocessor carries out those directives. Only text blocks selected by the preprocessor are compiled.

The preprocessor selects a single *text* item by evaluating the constant expression following each **#if** or **#elif** directive until it finds a true (nonzero) constant expression. It selects all text (including other preprocessor directives beginning with #) up to its associated **#elif**, **#else**, or **#endif**.

If all occurrences of *constant-expression* are false, or if no **#elif** directives appear, the preprocessor selects the text block after the **#else** clause. When there's no **#else** clause, and all instances of *constant-expression* in the **#if** block are false, no text block is selected.

The *constant-expression* is an integer constant expression with these additional restrictions:

- Expressions must have integral type and can include only integer constants, character constants, and the **defined** operator.
- The expression can't use `sizeof` or a type-cast operator.
- The target environment may be unable to represent all ranges of integers.

- The translation represents type `int` the same way as type `long`, and `unsigned int` the same way as `unsigned long`.
- The translator can translate character constants to a set of code values different from the set for the target environment. To determine the properties of the target environment, use an app built for that environment to check the values of the *LIMITS.H* macros.
- The expression must not query the environment, and must remain insulated from implementation details on the target computer.

Preprocessor operators

defined

The preprocessor operator **defined** can be used in special constant expressions, as shown by the following syntax:

> **defined(** *identifier* **)**
> **defined** *identifier*

This constant expression is considered true (nonzero) if the *identifier* is currently defined. Otherwise, the condition is false (0). An identifier defined as empty text is considered defined. The **defined** operator can be used in an **#if** and an **#elif** directive, but nowhere else.

In the following example, the **#if** and **#endif** directives control compilation of one of three function calls:

```
C

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

The function call to `credit` is compiled if the identifier `CREDIT` is defined. If the identifier `DEBIT` is defined, the function call to `debit` is compiled. If neither identifier is defined,

the call to `printerror` is compiled. Both `CREDIT` and `credit` are distinct identifiers in C and C++ because their cases are different.

The conditional compilation statements in the following example assume a previously defined symbolic constant named `DLEVEL`.

C

```
#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
```

The first **#if** block shows two sets of nested **#if**, **#else**, and **#endif** directives. The first set of directives is processed only if `DLEVEL > 5` is true. Otherwise, the statements after **#else** are processed.

The **#elif** and **#else** directives in the second example are used to make one of four choices, based on the value of `DLEVEL`. The constant `STACK` is set to 0, 100, or 200, depending on the definition of `DLEVEL`. If `DLEVEL` is greater than 5, then the statement

C

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

is compiled, and `STACK` isn't defined.

A common use for conditional compilation is to prevent multiple inclusions of the same header file. In C++, where classes are often defined in header files, constructs like this one can be used to prevent multiple definitions:

C++

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

class Example
{
    //...
};

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

The preceding code checks to see if the symbolic constant `EXAMPLE_H` is defined. If so, the file has already been included and doesn't need reprocessing. If not, the constant `EXAMPLE_H` is defined to mark EXAMPLE.H as already processed.

__has_include

Visual Studio 2017 version 15.3 and later: Determines whether a library header is available for inclusion:

C++

```
#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
```

See also

Preprocessor directives

`#ifdef` and `#ifndef` directives (C/C++)

Article • 08/03/2021

The `#ifdef` and `#ifndef` preprocessor directives have the same effect as the #if directive when it's used with the `defined` operator.

Syntax

> `#ifdef` *`identifier`*
> `#ifndef` *`identifier`*

These directives are equivalent to:

> `#if defined` *`identifier`*
> `#if !defined` *`identifier`*

Remarks

You can use the `#ifdef` and `#ifndef` directives anywhere `#if` can be used. The `#ifdef` *`identifier`* statement is equivalent to `#if 1` when *`identifier`* has been defined. It's equivalent to `#if 0` when *`identifier`* hasn't been defined, or has been undefined by the `#undef` directive. These directives check only for the presence or absence of identifiers defined with `#define`, not for identifiers declared in the C or C++ source code.

These directives are provided only for compatibility with previous versions of the language. The `defined(` *`identifier`* `)` constant expression used with the `#if` directive is preferred.

The `#ifndef` directive checks for the opposite of the condition checked by `#ifdef`. If the identifier hasn't been defined, or if its definition has been removed with `#undef`, the condition is true (nonzero). Otherwise, the condition is false (0).

Microsoft Specific

The *identifier* can be passed from the command line using the /D option. Up to 30 macros can be specified with `/D`.

The `#ifdef` directive is useful for checking whether a definition exists, because a definition can be passed from the command line. For example:

C++

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

END Microsoft Specific

See also

Preprocessor directives

#import directive (C++)

Article • 08/03/2021

C++ Specific

Used to incorporate information from a type library. The content of the type library is converted into C++ classes, mostly describing the COM interfaces.

Syntax

> **#import** "*filename*" [*attributes*]
> **#import** <*filename*> [*attributes*]

Parameters

filename
Specifies the type library to import. The *filename* can be one of the following kinds:

- The name of a file that contains a type library, such as an .olb, .tlb, or .dll file. The keyword, `file:`, can precede each filename.

- The progid of a control in the type library. The keyword, `progid:`, can precede each progid. For example:

  ```
  C++

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

  For more on progids, see Specifying the Localization ID and Version Number.

  When you use a 32-bit cross compiler on a 64-bit operating system, the compiler can only read the 32-bit registry hive. You might want to use the native 64-bit compiler to build and register a 64-bit type library.

- The library ID of the type library. The keyword, `libid:`, can precede each library ID. For example:

  ```
  C++

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

If you don't specify `version` or `lcid`, the rules applied to `progid:` are also applied to `libid:`.

- An executable (.exe) file.
- A library (.dll) file containing a type library resource (such as an .ocx).
- A compound document holding a type library.
- Any other file format that can be understood by the **LoadTypeLib** API.

attributes
One or more #import attributes. Separate attributes with either a space or comma. For example:

```
C++

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

-or-

```
C++

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

Remarks

Search order for filename

filename is optionally preceded by a directory specification. The file name must name an existing file. The difference between the two syntax forms is the order in which the preprocessor searches for the type library files when the path is incompletely specified.

Syntax form	Action
Quoted form	Instructs the preprocessor to look for type library files first in the directory of the file that contains the **#import** statement, and then in the directories of whatever files include (`#include`) that file. The preprocessor then searches along the paths shown below.

Syntax form	Action
Angle-bracket form	Instructs the preprocessor to search for type library files along the following paths: 1. The `PATH` environment variable path list 2. The `LIB` environment variable path list 3. The path specified by the /I compiler option, except it the compiler is searching for a type library that was referenced from another type library with the no_registry attribute.

Specify the localization ID and version number

When you specify a progid, you can also specify the localization ID and version number of the progid. For example:

```
C++

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

If you don't specify a localization ID, a progid is chosen according to the following rules:

- If there's only one localization ID, that one is used.
- If there's more than one localization ID, the first one with version number 0, 9, or 409 is used.
- If there's more than one localization ID and none of them are 0, 9, or 409, the last one is used.
- If you don't specify a version number, the most recent version is used.

Header files created by import

#import creates two header files that reconstruct the type library contents in C++ source code. The primary header file is similar to the one produced by the Microsoft Interface Definition Language (MIDL) compiler, but with additional compiler-generated code and data. The primary header file has the same base name as the type library, plus a .TLH extension. The secondary header file has the same base name as the type library, with a .TLI extension. It contains the implementations for compiler-generated member functions, and is included (`#include`) in the primary header file.

If importing a dispinterface property that uses `byref` parameters, **#import** doesn't generate a __declspec(property) statement for the function.

Both header files are placed in the output directory specified by the /Fo (name object file) option. They're then read and compiled by the compiler as if the primary header file was named by a `#include` directive.

The following compiler optimizations come with the **#import** directive:

- The header file, when created, is given the same timestamp as the type library.
- When **#import** is processed, the compiler first checks if the header exists and is up-to-date. If yes, then it doesn't need to be re-created.

The **#import** directive also participates in minimal rebuild and can be placed in a precompiled header file. For more information, see Creating precompiled header files.

Primary type library header file

The primary type library header file consists of seven sections:

- Heading boilerplate: Consists of comments, `#include` statement for COMDEF.H (which defines some standard macros used in the header), and other miscellaneous setup information.
- Forward references and typedefs: Consists of structure declarations such as `struct IMyInterface` and typedefs.
- Smart pointer declarations: The template class `_com_ptr_t` is a smart pointer. It encapsulates interface pointers, and eliminates the need to call `AddRef`, `Release`, and `QueryInterface` functions. It also hides the `CoCreateInstance` call when creating a new COM object. This section uses the macro statement `_COM_SMARTPTR_TYPEDEF` to establish typedefs of COM interfaces as template specializations of the _com_ptr_t template class. For example, for interface `IMyInterface`, the .TLH file will contain:

  TLH

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

  which the compiler will expand to:

  C++

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

Type `IMyInterfacePtr` can then be used in place of the raw interface pointer `IMyInterface*`. Consequently, there's no need to call the various `IUnknown` member functions

- Typeinfo declarations: Primarily consists of class definitions and other items exposing the individual typeinfo items returned by `ITypeLib:GetTypeInfo`. In this section, each typeinfo from the type library is reflected in the header in a form dependent on the `TYPEKIND` information.

- Optional old-style GUID definition: Contains initializations of the named GUID constants. These names have the form `CLSID_CoClass` and `IID_Interface`, similar to the ones generated by the MIDL compiler.

- `#include` statement for the secondary type library header.

- Footer boilerplate: Currently includes `#pragma pack(pop)`.

All sections, except the heading boilerplate and footer boilerplate section, are enclosed in a namespace with its name specified by the `library` statement in the original IDL file. You can use the names from the type library header by an explicit qualification using the namespace name. Or, you can include the following statement:

```
C++

using namespace MyLib;
```

immediately after the **#import** statement in the source code.

The namespace can be suppressed by using the no_namespace) attribute of the **#import** directive. However, suppressing the namespace may lead to name collisions. The namespace can also be renamed by the rename_namespace attribute.

The compiler provides the full path to any type library dependency required by the type library it's currently processing. The path is written, in the form of comments, into the type library header (.TLH) that the compiler generates for each processed type library.

If a type library includes references to types defined in other type libraries, then the .TLH file will include comments of the following sort:

```
TLH

//
// Cross-referenced type libraries:
//
```

```
//  #import "c:\path\typelib0.tlb"
//
```

The actual filename in the **#import** comment is the full path of the cross-referenced type library, as stored in the registry. If you encounter errors that are caused by missing type definitions, check the comments at the head of the .TLH to see which dependent type libraries may need to be imported first. Likely errors are syntax errors (for example, C2143, C2146, C2321), C2501 (missing decl-specifiers), or C2433 ('inline' not permitted on data declaration) while compiling the .TLI file.

To resolve dependency errors, determine which of the dependency comments aren't otherwise provided for by system headers, and then provide an **#import** directive at some point before the **#import** directive of the dependent type library.

#import attributes

#import can optionally include one or more attributes. These attributes tell the compiler to modify the contents of the type-library headers. A backslash (\) symbol can be used to include additional lines in a single **#import** statement. For example:

C++

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

For more information, see #import attributes.

END C++ Specific

See also

Preprocessor directives
Compiler COM support

#import attributes (C++)

Article • 08/03/2021

Provides links to attributes used with the `#import` directive.

Microsoft Specific

The following attributes are available to the `#import` directive.

Attribute	Description
auto_rename	Renames C++ reserved words by appending two underscores (__) to the variable name to resolve potential name conflicts.
auto_search	Specifies that, when a type library is referenced with #import and itself references another type library, the compiler can do an implicit #import for the other type library.
embedded_idl	Specifies that the type library is written to the .tlh file with the attribute-generated code preserved.
exclude	Excludes items from the type library header files being generated.
high_method_prefix	Specifies a prefix to be used in naming high-level properties and methods.
high_property_prefixes	Specifies alternate prefixes for three property methods.
implementation_only	Suppresses the generation of the .tlh header file (the primary header file).
include()	Disables automatic exclusion.
inject_statement	Inserts its argument as source text into the type-library header.
named_guids	Tells the compiler to define and initialize GUID variables in old style, of the form `LIBID_MyLib`, `CLSID_MyCoClass`, `IID_MyInterface`, and `DIID_MyDispInterface`.
no_auto_exclude	Disables automatic exclusion.
no_dual_interfaces	Changes the way the compiler generates wrapper functions for dual interface methods.
no_implementation	Suppresses the generation of the .tli header, which contains the implementations of the wrapper member functions.
no_namespace	Specifies that the namespace name is not generated by the compiler.

Attribute	Description
no_registry	Tells the compiler not to search the registry for type libraries.
no_search_namespace	Has the same functionality as the no_namespace attribute but is used on type libraries that you use the #import directive with the auto_search attribute.
no_smart_pointers	Suppresses the creation of smart pointers for all interfaces in the type library.
raw_dispinterfaces	Tells the compiler to generate low-level wrapper functions for dispinterface methods and properties that call `IDispatch::Invoke` and return the HRESULT error code.
raw_interfaces_only	Suppresses the generation of error-handling wrapper functions and property declarations that use those wrapper functions.
raw_method_prefix	Specifies a different prefix to avoid name collisions.
raw_native_types	Disables the use of COM support classes in the high-level wrapper functions and forces the use of low-level data types instead.
raw_property_prefixes	Specifies alternate prefixes for three property methods.
rename	Works around name collision problems.
rename_namespace	Renames the namespace that contains the contents of the type library.
rename_search_namespace	Has the same functionality as the rename_namespace attribute but is used on type libraries that you use the #import directive with the auto_search attribute.
tlbid	Allows for loading libraries other than the primary type library.

END Microsoft Specific

See also

#import directive

auto_rename import attribute

Article • 08/03/2021

C++ Specific

Renames C++ reserved words by appending two underscores (__) to the variable name to resolve potential name conflicts.

Syntax

> **#import** *type-library* **auto_rename**

Remarks

This attribute is used when importing a type library that uses one or more C++ reserved words (keywords or macros) as variable names.

END C++ Specific

See also

#import attributes
#import directive

auto_search import attribute

Article • 08/03/2021

C++ Specific

Specifies that, when a type library is referenced with `#import` and itself references another type library, the compiler can do an implicit `#import` for the other type library.

Syntax

> **#import** *type-library* **auto_search**

Remarks

END C++ Specific

See also

#import attributes
#import directive

embedded_idl import attribute

Article • 08/03/2021

C++ Specific

Specifies whether the type library is written to the `.tlh` file with the attribute-generated code preserved.

Syntax

> **#import** *type-library* **embedded_idl** [({ **"emitidl"** | **"no_emitidl"** })]

Parameters

"emitidl"
Type information imported from *type-library* is present in the IDL generated for the attributed project. This behavior is the default, and is in effect if you don't specify a parameter to `embedded_idl`.

"no_emitidl"
Type information imported from *type-library* isn't present in the IDL generated for the attributed project.

Example

C++

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

END C++ Specific

See also

#import attributes
#import directive

exclude import attribute

Article • 08/03/2021

C++ Specific

Excludes items from the type library header files being generated.

Syntax

> **#import** *type-library* **exclude(** "*Name1*" [**,** "*Name2*" ...] **)**

Parameters

Name1
First item to be excluded.

Name2
(Optional) Second and later items to be excluded, if necessary.

Remarks

Type libraries may include definitions of items defined in system headers or other type libraries. This attribute can take any number of arguments, where each is a top-level type library item to be excluded.

END C++ Specific

See also

#import attributes
#import directive

high_method_prefix import attribute

Article • 08/03/2021

C++ Specific

Specifies a prefix to be used in naming high-level properties and methods.

Syntax

> **#import** *type-library* **high_method_prefix(** "*Prefix*" **)**

Parameters

Prefix
Prefix to be used.

Remarks

By default, high-level error-handling properties and methods are exposed by member functions named without a prefix. The names are from the type library.

END C++ Specific

See also

#import attributes
#import directive

high_property_prefixes import attribute

Article • 08/03/2021

C++ Specific

Specifies alternate prefixes for three property methods.

Syntax

> **#import** *type-library* **high_property_prefixes(** "*GetPrefix*" **,** "*PutPrefix*" **,** "*PutRefPrefix*" **)**

Parameters

GetPrefix
Prefix to be used for the `propget` methods.

PutPrefix
Prefix to be used for the `propput` methods.

PutRefPrefix
Prefix to be used for the `propputref` methods.

Remarks

By default, high-level error-handling `propget`, `propput`, and `propputref` methods are exposed by member functions named with prefixes `Get`, `Put`, and `PutRef`, respectively.

END C++ Specific

See also

#import attributes
#import directive

implementation_only import attribute

Article • 08/03/2021

C++ Specific

Suppresses the generation of the `.tlh` primary type-library header file.

Syntax

> **#import** *type-library* **implementation_only**

Remarks

This file contains all the declarations used to expose the type-library contents. The `.tli` header file, with the implementations of the wrapper member functions, will be generated and included in the compilation.

When this attribute is specified, the content of the `.tli` header is in the same namespace as the one normally used in the `.tlh` header. In addition, the member functions are not declared as inline.

The **implementation_only** attribute is intended for use in conjunction with the no_implementation attribute as a way of keeping the implementations out of the precompiled header (PCH) file. An `#import` statement with the `no_implementation` attribute is placed in the source region used to create the PCH. The resulting PCH is used by a number of source files. An `#import` statement with the **implementation_only** attribute is then used outside the PCH region. You're required to use this statement only once in one of the source files. It generates all the required wrapper member functions without additional recompilation for each source file.

> ⓘ **Note**
>
> The **implementation_only** attribute in one `#import` statement must be used in conjunction with another `#import` statement, of the same type library, with the `no_implementation` attribute. Otherwise, compiler errors are generated. This is because wrapper class definitions generated by the `#import` statement with the `no_implementation` attribute are required to compile the implementations generated by the **implementation_only** attribute.

END C++ Specific

See also

#import attributes
#import directive

include() import attribute

Article • 08/03/2021

C++ Specific

Disables automatic exclusion.

Syntax

> **#import** *type-library* **include(** "*Name1*" [,"*Name2*" ...] **)**

Parameters

Name1
First item to be forcibly included.

Name2
Second item to be forcibly included (if necessary).

Remarks

Type libraries may include definitions of items defined in system headers or other type libraries. `#import` attempts to avoid multiple definition errors by automatically excluding such items. If some items shouldn't be excluded automatically, you may see Compiler Warning (level 3) C4192. You can use this attribute to disable the automatic exclusion. This attribute can take any number of arguments, one for each name of a type-library item to be included.

END C++ Specific

See also

#import attributes
#import directive

inject_statement import attribute

Article • 08/03/2021

C++ Specific

Inserts its argument as source text into the type-library header.

Syntax

> **#import** *type-library* **inject_statement(** "*source-text*" **)**

Parameters

source-text
Source text to be inserted into the type library header file.

Remarks

The text is placed at the beginning of the namespace declaration that wraps the *type-library* contents in the header file.

END C++ Specific

See also

#import attributes
#import directive

named_guids import attribute

Article • 08/03/2021

C++ Specific

Tells the compiler to define and initialize GUID variables in the old style, of the form `LIBID_MyLib`, `CLSID_MyCoClass`, `IID_MyInterface`, and `DIID_MyDispInterface`.

Syntax

> **#import** *type-library* **named_guids**

END C++ Specific

See also

#import attributes
#import directive

no_auto_exclude import attribute

Article • 08/03/2021

C++ Specific

Disables automatic exclusion.

Syntax

> **#import** *type-library* **no_auto_exclude**

Remarks

Type libraries may include definitions of items defined in system headers or other type libraries. `#import` attempts to avoid multiple definition errors by automatically excluding such items. It causes Compiler Warning (level 3) C4192 to be issued for each item to be excluded. You can disable the automatic exclusion by using this attribute.

END C++ Specific

See also

#import attributes
#import directive

no_dual_interfaces import attribute

Article • 08/03/2021

C++ Specific

Changes the way the compiler generates wrapper functions for dual interface methods.

Syntax

> **#import** *type-library* **no_dual_interfaces**

Remarks

Normally, the wrapper calls the method through the virtual function table for the interface. With **no_dual_interfaces**, the wrapper instead calls `IDispatch::Invoke` to invoke the method.

END C++ Specific

See also

#import attributes
#import directive

no_implementation import attribute

Article • 08/03/2021

C++ Specific

Suppresses the generation of the `.tli` header, which contains the implementations of the wrapper member functions.

Syntax

> **#import** *type-library* **no_implementation**

Remarks

If this attribute is specified, the `.tlh` header, with the declarations to expose type-library items, will be generated without an `#include` statement to include the `.tli` header file.

This attribute is used in conjunction with implementation_only.

END C++ Specific

See also

#import attributes
#import directive

no_namespace import attribute

Article • 08/03/2021

C++ Specific

Specifies that the compiler doesn't generate a namespace name.

Syntax

> **#import** *type-library* **no_namespace**

Remarks

The type-library contents in the `#import` header file are normally defined in a namespace. The namespace name is specified in the `library` statement of the original IDL file. If the **no_namespace** attribute is specified, then this namespace isn't generated by the compiler.

If you want to use a different namespace name, then use the rename_namespace attribute instead.

END C++ Specific

See also

#import attributes
#import directive

no_registry import attribute

Article • 08/03/2021

no_registry tells the compiler not to search the registry for type libraries imported with `#import`.

Syntax

> **#import** *type-library* **no_registry**

Parameters

type-library
A type library.

Remarks

If a referenced type library isn't found in the include directories, the compilation fails even if the type library is in the registry. **no_registry** propagates to other type libraries implicitly imported with `auto_search`.

The compiler never searches the registry for type libraries that are specified by file name and passed directly to `#import`.

When `auto_search` is specified, the additional `#import` directives are generated by using the **no_registry** setting of the initial `#import`. If the initial `#import` directive was **no_registry**, an `auto_search`-generated `#import` is also **no_registry**.

no_registry is useful if you want to import cross-referenced type libraries. It keeps the compiler from finding an older version of the file in the registry. **no_registry** is also useful if the type library isn't registered.

See also

#import attributes
#import directive

no_search_namespace import attribute

Article • 08/03/2021

C++ Specific

Has the same functionality as the no_namespace attribute, but is used on type libraries where you use the `#import` directive with the auto_search attribute.

Syntax

> **#import** *type-library* **no_search_namespace**

END C++ Specific

See also

#import attributes
#import directive

no_smart_pointers import attribute

Article • 08/03/2021

C++ Specific

Suppresses the creation of smart pointers for all interfaces in the type library.

Syntax

> **#import** *type-library* **no_smart_pointers**

Remarks

By default, when you use `#import`, you get a smart pointer declaration for all interfaces in the type library. These smart pointers are of type _com_ptr_t.

END C++ Specific

See also

#import attributes
#import directive

raw_dispinterfaces import attribute

Article • 08/03/2021

C++ Specific

Tells the compiler to generate low-level wrapper functions for dispinterface methods, and for properties that call `IDispatch::Invoke` and return the HRESULT error code.

Syntax

> **#import** *type-library* **raw_dispinterfaces**

Remarks

If this attribute isn't specified, only high-level wrappers are generated, which throw C++ exceptions on failure.

END C++ Specific

See also

#import attributes
#import directive

raw_interfaces_only import attribute

Article • 08/03/2021

C++ Specific

Suppresses the generation of error-handling wrapper functions, and property declarations that use those wrapper functions.

Syntax

> **#import** *type-library* **raw_interfaces_only**

Remarks

The **raw_interfaces_only** attribute also causes the default prefix used in naming the non-property functions to be removed. Normally, the prefix is `raw_`. If this attribute is specified, the function names are taken directly from the type library.

This attribute allows you to expose only the low-level contents of the type library.

END C++ Specific

See also

#import attributes
#import directive

raw_method_prefix

Article • 08/03/2021

C++ Specific

Specifies a different prefix to avoid name collisions.

Syntax

> **#import** *type-library* **raw_method_prefix(** "*Prefix*" **)**

Parameters

Prefix
The prefix to be used.

Remarks

Low-level properties and methods are exposed by member functions named using a default prefix of **raw_**, to avoid name collisions with the high-level error-handling member functions.

> **Note**
>
> The effects of the **raw_method_prefix** attribute are unchanged by the presence of the raw_interfaces_only attribute. The **raw_method_prefix** always takes precedence over `raw_interfaces_only` in specifying a prefix. If both attributes are used in the same `#import` statement, then the prefix specified by the **raw_method_prefix** attribute is used.

END C++ Specific

See also

#import attributes
#import directive

raw_native_types import attribute

Article • 08/03/2021

C++ Specific

Disables the use of COM support classes in the high-level wrapper functions, and forces the use of low-level data types instead.

Syntax

> **#import** *type-library* **raw_native_types**

Remarks

By default, the high-level error-handling methods use the COM support classes _bstr_t and _variant_t in place of the `BSTR` and `VARIANT` data types and raw COM interface pointers. These classes encapsulate the details of allocating and deallocating memory storage for these data types, and greatly simplify type casting and conversion operations.

END C++ Specific

See also

#import attributes
#import directive

raw_property_prefixes import attribute

Article • 08/03/2021

C++ Specific

Specifies alternate prefixes for three property methods.

Syntax

> **#import** *type-library* **raw_property_prefixes(** "*GetPrefix*" **,** "*PutPrefix*" **,** "*PutRefPrefix*" **)**

Parameters

GetPrefix
Prefix to use for the `propget` methods.

PutPrefix
Prefix to use for the `propput` methods.

PutRefPrefix
Prefix to use for the `propputref` methods.

Remarks

By default, low-level `propget`, `propput`, and `propputref` methods are exposed by member functions named using prefixes of `get_`, `put_`, and `putref_`, respectively. These prefixes are compatible with the names used in the header files generated by MIDL.

END C++ Specific

See also

#import attributes
#import directive

rename import attribute

Article • 08/03/2021

C++ Specific

Works around name collision problems.

Syntax

> **#import** *type-library* **rename(** "*OldName*" **,** "*NewName*" **)**

Parameters

OldName
Old name in the type library.

NewName
Name to be used instead of the old name.

Remarks

When the **rename** attribute is specified, the compiler replaces all occurrences of *OldName* in *type-library* with the user-supplied *NewName* in the resulting header files.

The **rename** attribute can be used when a name in the type library coincides with a macro definition in the system header files. If this situation isn't resolved, the compiler may issue various syntax errors, such as Compiler Error C2059 and Compiler Error C2061.

> **Note**
>
> The replacement is for a name used in the type library, not for a name used in the resulting header file.

For example, suppose a property named `MyParent` exists in a type library, and a macro `GetMyParent` is defined in a header file and used before `#import`. Since `GetMyParent` is the default name of a wrapper function for the error-handling `get` property, a name collision will occur. To work around the problem, use the following attribute in the `#import` statement:

```
C++
```

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

which renames the name `MyParent` in the type library. An attempt to rename the `GetMyParent` wrapper name will fail:

```
C++
```

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

It's because the name `GetMyParent` only occurs in the resulting type library header file.

END C++ Specific

See also

#import attributes
#import directive

rename_namespace import attribute

Article • 08/03/2021

C++ Specific

Renames the namespace that contains the contents of the type library.

Syntax

> **#import** *type-library* **rename_namespace(** "*NewName*" **)**

Parameters

NewName
The new name of the namespace.

Remarks

The **rename_namespace** attribute takes a single argument, *NewName*, which specifies the new name for the namespace.

To remove the namespace, use the no_namespace attribute instead.

END C++ Specific

See also

#import attributes
#import directive

rename_search_namespace import attribute

Article • 08/03/2021

C++ Specific

Has the same functionality as the rename_namespace attribute, but is used on type libraries where you use the `#import` directive along with the auto_search attribute.

Syntax

> **#import** *type-library* **rename_search_namespace(** "*NewName*" **)**

Parameters

NewName
The new name of the namespace.

Remarks

END C++ Specific

See also

#import attributes
#import directive

tlbid import attribute

Article • 08/03/2021

C++ Specific

Allows for loading libraries other than the primary type library.

Syntax

> **#import** *type-library-dll* **tlbid(** *number* **)**

Parameters

number
The number of the type library in *type-library-dll*.

Remarks

If multiple type libraries are built into a single DLL, it's possible to load libraries other than the primary type library by using **tlbid**.

For example:

```
C++

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

is equivalent to:

```
C++

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

END C++ Specific

See also

#import attributes
#import directive

`#include` directive (C/C++)

Article • 02/18/2022

Tells the preprocessor to include the contents of a specified file at the point where the directive appears.

Syntax

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

Remarks

You can organize constant and macro definitions into *include files* (also known as *header files*) and then use `#include` directives to add them to any source file. Include files are also useful for incorporating declarations of external variables and complex data types. The types may be defined and named only once in an include file created for that purpose.

The *path-spec* is a file name that may optionally be preceded by a directory specification. The file name must name an existing file. The syntax of the *path-spec* depends on the operating system on which the program is compiled.

For information about how to reference assemblies in a C++ application that's compiled by using /clr, see #using directive.

Both syntax forms cause the `#include` directive to be replaced by the entire contents of the specified file. The difference between the two forms is the order of the paths that the preprocessor searches when the path is incompletely specified. The following table shows the difference between the two syntax forms.

Syntax Form	Action

Syntax Form	Action
Quoted form	The preprocessor searches for include files in this order: 1) In the same directory as the file that contains the `#include` statement. 2) In the directories of the currently opened include files, in the reverse order in which they were opened. The search begins in the directory of the parent include file and continues upward through the directories of any grandparent include files. 3) Along the path that's specified by each `/I` compiler option. 4) Along the paths that are specified by the `INCLUDE` environment variable.
Angle-bracket form	The preprocessor searches for include files in this order: 1) Along the path that's specified by each `/I` compiler option. 2) When compiling occurs on the command line, along the paths that are specified by the `INCLUDE` environment variable.

The preprocessor stops searching as soon as it finds a file that has the given name. If you enclose a complete, unambiguous path specification for the include file between double quotation marks (`" "`), the preprocessor searches only that path specification and ignores the standard directories.

If the file name that's enclosed in double quotation marks is an incomplete path specification, the preprocessor first searches the *parent* file's directory. A parent file is the file that contains the `#include` directive. For example, if you include a file named *file2* in a file named *file1*, *file1* is the parent file.

Include files can be *nested*: An `#include` directive can appear in a file that's named by another `#include` directive. For example, *file2* could include *file3*. In this case, *file1* would still be the parent of *file2*, but it would be the grandparent of *file3*.

When include files are nested and when compiling occurs on the command line, directory searching begins in the directory of the parent file. Then it proceeds through the directories of any grandparent files. That is, searching begins relative to the directory that contains the source that's currently being processed. If the file isn't found, the search moves to directories that are specified by the /I (Additional include directories) compiler option. Finally, the directories that are specified by the `INCLUDE` environment variable are searched.

Within the Visual Studio development environment, the `INCLUDE` environment variable is ignored. The values specified in the project properties for include directories are used instead. For more information about how to set the include directories in Visual Studio, see Include Directories and Additional Include Directories.

This example shows file inclusion by using angle brackets:

C

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

The example adds the contents of the file named *`stdio.h`* to the source program. The angle brackets cause the preprocessor to search the directories that are specified by the `INCLUDE` environment variable for *`stdio.h`*, after it searches directories that are specified by the `/I` compiler option.

The next example shows file inclusion by using the quoted form:

C

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

The example adds the contents of the file that's specified by *`defs.h`* to the source program. The quotation marks mean that the preprocessor first searches the directory that contains the parent source file.

Nesting of include files can continue up to 10 levels. When processing of the nested `#include` is finished, the preprocessor continues to insert the enclosing parent include file into the original source file.

Microsoft-specific

To locate the source files to include, the preprocessor first searches the directories specified by the `/I` compiler option. If the `/I` option isn't present, or if it fails, the preprocessor uses the `INCLUDE` environment variable to find any include files within angle brackets. The `INCLUDE` environment variable and `/I` compiler option can contain multiple paths, separated by semicolons (`;`). If more than one directory appears as part of the `/I` option or within the `INCLUDE` environment variable, the preprocessor searches them in the order in which they appear.

For example, the command

Windows Command Prompt

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

causes the preprocessor to search the directory `D:\msvc\include\` for include files such as `stdio.h`. The commands

Windows Command Prompt

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

have the same effect. If both sets of searches fail, a fatal compiler error is generated.

If the file name is fully specified for an include file that has a path that includes a colon (for example, `F:\MSVC\SPECIAL\INCL\TEST.H`), the preprocessor follows the path.

For include files that are specified as `#include "path-spec"`, directory search begins in the directory of the parent file and then proceeds through the directories of any grandparent files. That is, the search begins relative to the directory that contains the source file that's being processed. If there's no grandparent file and the file still isn't found, the search continues as if the file name were enclosed in angle brackets.

END Microsoft-specific

See also

Preprocessor directives
/I (Additional include directories)

#line directive (C/C++)

Article • 03/14/2022

The **#line** directive tells the preprocessor to set the compiler's reported values for the line number and filename to a given line number and filename.

Syntax

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

Remarks

The compiler uses the line number and optional filename to refer to errors that it finds during compilation. The line number usually refers to the current input line, and the filename refers to the current input file. The line number is incremented after each line is processed.

The *digit-sequence* value can be any integer constant within the range from 0 to 2147483647, inclusive. Macro replacement can be used on the preprocessing tokens, but the result must evaluate to the correct syntax. The *filename* can be any combination of characters and must be enclosed in double quotation marks (`" "`). If *filename* is omitted, the previous filename remains unchanged.

You can alter the source line number and filename by writing a `#line` directive. The `#line` directive sets the value for the line that immediately follows the directive in the source file. The translator uses the line number and filename to determine the values of the predefined macros `__FILE__` and `__LINE__`. You can use these macros to insert self-descriptive error messages into the program text. For more information on these predefined macros, see Predefined macros.

The `__FILE__` macro expands to a string whose contents are the filename, surrounded by double quotation marks (`" "`).

If you change the line number and filename, the compiler ignores the previous values and continues processing with the new values. The **#line** directive is typically used by program generators. It's used to cause error messages to refer to the original source file, instead of to the generated program.

Example

The following examples illustrate `#line` and the `__LINE__` and `__FILE__` macros.

In the first example, the line number is set to 10, then to 20, and the filename is changed to *hello.cpp*.

C++

```
// 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__ );
}
```

Output

```
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
```

In this example, the macro `ASSERT` uses the predefined macros `__LINE__` and `__FILE__` to print an error message about the source file if a given assertion isn't true.

C

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

See also

Preprocessor directives

Null directive

Article • 08/03/2021

The null preprocessor directive is a single number sign (#) alone on a line. It has no effect.

Syntax

```
#
```

See also

Preprocessor directives

#undef directive (C/C++)

Article • 08/03/2021

Removes (undefines) a name previously created with `#define`.

Syntax

> **#undef** *identifier*

Remarks

The **#undef** directive removes the current definition of *identifier*. Consequently, subsequent occurrences of *identifier* are ignored by the preprocessor. To remove a macro definition using **#undef**, give only the macro *identifier*, not a parameter list.

You can also apply the **#undef** directive to an identifier that has no previous definition. This ensures that the identifier is undefined. Macro replacement isn't performed within **#undef** statements.

The **#undef** directive is typically paired with a `#define` directive to create a region in a source program in which an identifier has a special meaning. For example, a specific function of the source program can use manifest constants to define environment-specific values that do not affect the rest of the program. The **#undef** directive also works with the `#if` directive to control conditional compilation of the source program. For more information, see The #if, #elif, #else, and #endif directives.

In the following example, the **#undef** directive removes definitions of a symbolic constant and a macro. Note that only the identifier of the macro is given.

C

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

Microsoft Specific

Macros can be undefined from the command line using the `/U` option, followed by the macro names to be undefined. The effect of issuing this command is equivalent to a sequence of `#undef` *macro-name* statements at the beginning of the file.

END Microsoft Specific

See also

Preprocessor directives

#using directive (C++/CLI)

Article • 06/30/2022

Imports metadata into a program compiled with /clr.

Syntax

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

Parameters

file
A Microsoft intermediate language (MSIL) `.dll`, `.exe`, `.netmodule`, or `.obj` file. For example,

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

`as_friend`
Specifies that all types in *file* are accessible. For more information, see Friend Assemblies (C++).

Remarks

file can be a Microsoft intermediate language (MSIL) file that you import for its managed data and managed constructs. If a DLL contains an assembly manifest, then all the DLLs referenced in the manifest are imported. The assembly you're building will list *file* in the metadata as an assembly reference.

Perhaps *file* doesn't contain an assembly (*file* is a module), and you don't intend to use type information from the module in the current (assembly) application. You may indicate the module is part of the assembly by using /ASSEMBLYMODULE. The types in the module would then be available to any application that referenced the assembly.

An alternative to use **`#using`** is the /FU compiler option.

.exe assemblies passed to **`#using`** should be compiled by using one of the .NET Visual Studio compilers (Visual Basic or Visual C#, for example). Attempting to import metadata from an .exe assembly compiled with **`/clr`** will result in a file load exception.

> ⓘ Note

A component that is referenced with `#using` can be run with a different version of the file imported at compile time, causing a client application to give unexpected results.

In order for the compiler to recognize a type in an assembly (not a module), it needs to be forced to resolve the type. You can force it, for example, by defining an instance of the type. There are other ways to resolve type names in an assembly for the compiler. For example, if you inherit from a type in an assembly, the type name becomes known to the compiler.

When importing metadata built from source code that used __declspec(thread), the thread semantics aren't persisted in metadata. For example, a variable declared with `__declspec(thread)`, compiled in a program that is built for the .NET Framework common language runtime, and then imported via `#using`, won't have `__declspec(thread)` semantics on the variable.

All imported types (both managed and native) in a file referenced by `#using` are available, but the compiler treats native types as declarations, not definitions.

mscorlib.dll is automatically referenced when compiling with `/clr`.

The LIBPATH environment variable specifies the directories to search when the compiler resolves file names passed to `#using`.

The compiler searches for references along the following path:

- A path specified in the `#using` statement.
- The current directory.
- The .NET Framework system directory.
- Directories added with the /AI compiler option.
- Directories on LIBPATH environment variable.

Examples

You can build an assembly that references a second assembly that itself references a third assembly. You only have to explicitly reference the third assembly from the first one if you explicitly use one of its types.

Source file `using_assembly_A.cpp`:

C++

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

Source file `using_assembly_B.cpp`:

C++

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

In the following sample, there's the compiler doesn't report an error about referencing *using_assembly_A.dll*, because the program doesn't use any of the types defined in *using_assembly_A.cpp*.

C++

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

See also

Preprocessor directives

Preprocessor operators

Article • 08/03/2021

Four preprocessor-specific operators are used in the context of the `#define` directive. See the following table for a summary of each. The stringizing, charizing, and token-pasting operators are discussed in the next three sections. For information on the `defined` operator, see The #if, #elif, #else, and #endif directives.

Operator	Action
Stringizing operator (#)	Causes the corresponding actual argument to be enclosed in double quotation marks
Charizing operator (#@)	Causes the corresponding argument to be enclosed in single quotation marks and to be treated as a character (Microsoft-specific)
Token-pasting operator (##)	Allows tokens used as actual arguments to be concatenated to form other tokens
defined operator	Simplifies the writing of compound expressions in certain macro directives

See also

Preprocessor directives
Predefined macros
c/c++ preprocessor reference

Stringizing operator (#)

Article • 08/03/2021

The number-sign or "stringizing" operator (#) converts macro parameters to string literals without expanding the parameter definition. It's used only with macros that take arguments. If it precedes a formal parameter in the macro definition, the actual argument passed by the macro invocation is enclosed in quotation marks and treated as a string literal. The string literal then replaces each occurrence of a combination of the stringizing operator and formal parameter within the macro definition.

> **Note**
>
> The Microsoft C (versions 6.0 and earlier) extension to the ANSI C standard that previously expanded macro formal arguments appearing inside string literals and character constants is no longer supported. Code that relied on this extension should be rewritten using the stringizing (#) operator.

White space that precedes the first token and follows the last token of the actual argument is ignored. Any white space between the tokens in the actual argument is reduced to a single white space in the resulting string literal. Thus, if a comment occurs between two tokens in the actual argument, it's reduced to a single white space. The resulting string literal is automatically concatenated with any adjacent string literals that are separated only by white space.

Further, if a character contained in the argument usually requires an escape sequence when used in a string literal, for example, the quotation mark (`"`) or backslash (`\`) character, the necessary escape backslash is automatically inserted before the character.

The Microsoft C++ stringizing operator doesn't behave correctly when it's used with strings that include escape sequences. In this situation, the compiler generates Compiler Error C2017.

Examples

The following example shows a macro definition that includes the stringizing operator, and a main function that invokes the macro:

C++

```
// 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" );
}
```

The `stringer` macros are expanded during preprocessing, producing the following code:

C++

```
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" );
}
```

Output

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

The following sample shows how you can expand a macro parameter:

C++

```
// 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)
```

See also

Preprocessor operators

Charizing operator (#@)

Article • 08/03/2021

Microsoft Specific

The charizing operator can be used only with arguments of macros. If `#@` precedes a formal parameter in the definition of the macro, the actual argument is enclosed in single quotation marks and treated as a character when the macro is expanded. For example:

```
C++
```

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

causes the statement

```
C++
```

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

to be expanded to

```
C++
```

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

The single-quotation character (`'`) can't be used with the charizing operator.

END Microsoft Specific

See also

Preprocessor operators

Token-pasting operator (##)

Article • 08/03/2021

The double-number-sign or *token-pasting* operator (##), which is sometimes called the *merging* or *combining* operator, is used in both object-like and function-like macros. It permits separate tokens to be joined into a single token, and therefore, can't be the first or last token in the macro definition.

If a formal parameter in a macro definition is preceded or followed by the token-pasting operator, the formal parameter is immediately replaced by the unexpanded actual argument. Macro expansion is not performed on the argument prior to replacement.

Then, each occurrence of the token-pasting operator in *token-string* is removed, and the tokens preceding and following it are concatenated. The resulting token must be a valid token. If it is, the token is scanned for possible replacement if it represents a macro name. The identifier represents the name by which the concatenated tokens will be known in the program before replacement. Each token represents a token defined elsewhere, either within the program or on the compiler command line. White space preceding or following the operator is optional.

This example illustrates use of both the stringizing and token-pasting operators in specifying program output:

C++

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

If a macro is called with a numeric argument like

C++

```
paster( 9 );
```

the macro yields

C++

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

which becomes

```
```

C++

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

Example

C++

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

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

Output

```
token9 = 9
```

See also

Preprocessor operators

Macros (C/C++)

Article • 08/03/2021

The preprocessor expands macros in all lines except *preprocessor directives*, lines that have a # as the first non-white-space character. It expands macros in parts of some directives that aren't skipped as part of a conditional compilation. *Conditional compilation directives* allow you to suppress compilation of parts of a source file. They test a constant expression or identifier to determine which text blocks to pass on to the compiler, and which ones to remove from the source file during preprocessing.

The `#define` directive is typically used to associate meaningful identifiers with constants, keywords, and commonly used statements or expressions. Identifiers that represent constants are sometimes called *symbolic constants* or *manifest constants*. Identifiers that represent statements or expressions are called *macros*. In this preprocessor documentation, only the term "macro" is used.

When the name of a macro is recognized in the program source text, or in the arguments of certain other preprocessor commands, it's treated as a call to that macro. The macro name is replaced by a copy of the macro body. If the macro accepts arguments, the actual arguments following the macro name are substituted for formal parameters in the macro body. The process of replacing a macro call with the processed copy of the body is called *expansion* of the macro call.

In practical terms, there are two types of macros. *Object-like* macros take no arguments. *Function-like* macros can be defined to accept arguments, so that they look and act like function calls. Because macros don't generate actual function calls, you can sometimes make programs run faster by replacing function calls with macros. (In C++, inline functions are often a preferred method.) However, macros can create problems if you don't define and use them with care. You may have to use parentheses in macro definitions with arguments to preserve the proper precedence in an expression. Also, macros may not correctly handle expressions with side effects. For more information, see the `getrandom` example in The #define directive.

Once you've defined a macro, you can't redefine it to a different value without first removing the original definition. However, you can redefine the macro with exactly the same definition. Thus, the same definition may appear more than once in a program.

The `#undef` directive removes the definition of a macro. Once you've removed the definition, you can redefine the macro to a different value. The #define directive and The #undef directive discuss the `#define` and `#undef` directives, respectively.

For more information, see,

- Macros and C++
- Variadic Macros
- Predefined macros

See also

C/C++ preprocessor reference

Macros and C++

Article • 08/03/2021

C++ offers new capabilities, some of which supplant the ones offered by the ANSI C preprocessor. These new capabilities enhance the type safety and predictability of the language:

- In C++, objects declared as `const` can be used in constant expressions. It allows programs to declare constants that have type and value information. They can declare enumerations that can be viewed symbolically with the debugger. When you use the preprocessor `#define` directive to define constants, it's not as precise, and not type-safe. No storage is allocated for a `const` object, unless the program contains an expression that takes its address.

- The C++ inline function capability supplants function-type macros. The advantages of using inline functions over macros are:

  - Type safety. Inline functions are subject to the same type checking as normal functions. Macros aren't type-safe.

  - Correct handling of arguments that have side effects. Inline functions evaluate the expressions supplied as arguments before the function body is entered. Therefore, there's no chance that an expression with side effects will be unsafe.

For more information on inline functions, see inline, __inline, __forceinline.

For backward compatibility, all preprocessor facilities that existed in ANSI C and in earlier C++ specifications are preserved for Microsoft C++.

See also

Predefined macros
Macros (C/C++)

Variadic macros

Article • 08/03/2021

Variadic macros are function-like macros that contain a variable number of arguments.

Remarks

To use variadic macros, the ellipsis may be specified as the final formal argument in a macro definition, and the replacement identifier `__VA_ARGS__` may be used in the definition to insert the extra arguments. `__VA_ARGS__` is replaced by all of the arguments that match the ellipsis, including commas between them.

The C Standard specifies that at least one argument must be passed to the ellipsis to ensure the macro doesn't resolve to an expression with a trailing comma. The traditional Microsoft C++ implementation suppresses a trailing comma if no arguments are passed to the ellipsis. When the /Zc:preprocessor compiler option is set, the trailing comma isn't suppressed.

Example

C++

```
// 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
}
```

Output

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

See also

Macros (C/C++)

Predefined macros

Article • 06/11/2024

The Microsoft C/C++ compiler (MSVC) predefines certain preprocessor macros depending on the language (C or C++), the compilation target, and the chosen compiler options.

MSVC supports the predefined preprocessor macros required by the ANSI/ISO C99, C11, and C17 standards, and the ISO C++14, C++17, and C++20 standards. The implementation also supports several more Microsoft-specific preprocessor macros.

Some macros are defined only for specific build environments or compiler options. Except where noted, the macros are defined throughout a translation unit as if they were specified as `/D` compiler option arguments. When defined, the preprocessor expands macros their specified values before compilation. The predefined macros take no arguments and can't be redefined.

Standard predefined identifier

The compiler supports this predefined identifier specified by ISO C99 and ISO C++11.

- `__func__` The unqualified and unadorned name of the enclosing function as a function-local **static const** array of `char`.

  C++

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

Standard predefined macros

The compiler supports these predefined macros specified by the ISO C99, C11, C17, and ISO C++17 standards.

- `__cplusplus` Defined as an integer literal value when the translation unit is compiled as C++. Otherwise, undefined.
- `__DATE__` The compilation date of the current source file. The date is a constant length string literal of the form *Mmm dd yyyy*. The month name *Mmm* is the same

as the abbreviated month name generated by the C Runtime Library (CRT) asctime function. The first character of date *dd* is a space if the value is less than 10. This macro is always defined.

- `__FILE__` The name of the current source file. `__FILE__` expands to a character string literal. To ensure that the full path to the file is displayed, use `/FC` (Full Path of Source Code File in Diagnostics). This macro is always defined.

- `__LINE__` Defined as the integer line number in the current source file. The value of this macro can be changed by using a `#line` directive. The integral type of the value of `__LINE__` can vary depending on context. This macro is always defined.

- `__STDC__` Defined as 1 when compiled as C and if the /Za compiler option is specified. Starting in Visual Studio 2022 version 17.2, it's defined as 1 when compiled as C and if the /std:c11 or /std:c17 compiler option is specified. Otherwise, undefined.

- `__STDC_HOSTED__` Defined as 1 if the implementation is a *hosted implementation*, one that supports the entire required standard library. Otherwise, defined as 0.

- `__STDC_NO_ATOMICS__` Defined as 1 if the implementation doesn't support optional standard atomics. The MSVC implementation defines it as 1 when compiled as C and one of the /std C11 or C17 options is specified.

- `__STDC_NO_COMPLEX__` Defined as 1 if the implementation doesn't support optional standard complex numbers. The MSVC implementation defines it as 1 when compiled as C and one of the /std C11 or C17 options is specified.

- `__STDC_NO_THREADS__` Defined as 1 if the implementation doesn't support optional standard threads. The MSVC implementation defines it as 1 when compiled as C and one of the /std C11 or C17 options is specified.

- `__STDC_NO_VLA__` Defined as 1 if the implementation doesn't support standard variable length arrays. The MSVC implementation defines it as 1 when compiled as C and one of the /std C11 or C17 options is specified.

- `__STDC_VERSION__` Defined when compiled as C and one of the `/std` C11 or C17 options is specified. It expands to `201112L` for /std:c11, and `201710L` for /std:c17.

- `__STDCPP_DEFAULT_NEW_ALIGNMENT__` When /std:c17 or later is specified, this macro expands to a `size_t` literal that has the value of the alignment guaranteed by a call to alignment-unaware `operator new`. Larger alignments are passed to an

alignment-aware overload, such as `operator new(std::size_t, std::align_val_t)`. For more information, see /Zc:alignedNew (C++17 over-aligned allocation).

- `__STDCPP_THREADS__` Defined as 1 if and only if a program can have more than one thread of execution, and compiled as C++. Otherwise, undefined.
- `__TIME__` The time of translation of the preprocessed translation unit. The time is a character string literal of the form *hh:mm:ss*, the same as the time returned by the CRT asctime function. This macro is always defined.

Microsoft-specific predefined macros

MSVC supports other predefined macros:

- `__ARM_ARCH` Defined as an integer literal that represents the ARM architecture version. The value is defined as 8 for the Armv8-A architecture. For 8.1 and onwards, the value is scaled for minor versions, such as X.Y, by using the formula X * 100 + Y as defined by the ARM C language extension. For example, for Armv8.1, `__ARM_ARCH` is 8 * 100 + 1 or 801. To set the ARM architecture version, see /arch (ARM64). This macro was introduced in Visual Studio 2022 version 17.10.
- `__ATOM__` Defined as 1 when the /favor:ATOM compiler option is set and the compiler target is x86 or x64. Otherwise, undefined.
- `__AVX__` Defined as 1 when the /arch:AVX, /arch:AVX2, or /arch:AVX512 compiler options are set and the compiler target is x86 or x64. Otherwise, undefined.
- `__AVX2__` Defined as 1 when the /arch:AVX2 or /arch:AVX512 compiler option is set and the compiler target is x86 or x64. Otherwise, undefined.
- `__AVX512BW__` Defined as 1 when the /arch:AVX512 compiler option is set and the compiler target is x86 or x64. Otherwise, undefined.
- `__AVX512CD__` Defined as 1 when the /arch:AVX512 compiler option is set and the compiler target is x86 or x64. Otherwise, undefined.
- `__AVX512DQ__` Defined as 1 when the /arch:AVX512 compiler option is set and the compiler target is x86 or x64. Otherwise, undefined.
- `__AVX512F__` Defined as 1 when the /arch:AVX512 compiler option is set and the compiler target is x86 or x64. Otherwise, undefined.

- `__AVX512VL__` Defined as 1 when the /arch:AVX512 compiler option is set and the compiler target is x86 or x64. Otherwise, undefined.

- `_CHAR_UNSIGNED` Defined as 1 if the default `char` type is unsigned. This value is defined when the `/J` (Default char type is unsigned) compiler option is set. Otherwise, undefined.

- `__CLR_VER` Defined as an integer literal that represents the version of the Common Language Runtime (CLR) used to compile the app. The value is encoded in the form `Mmmbbbbb`, where `M` is the major version of the runtime, `mm` is the minor version of the runtime, and `bbbbb` is the build number. `__CLR_VER` is defined if the /clr compiler option is set. Otherwise, undefined.

  C++

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

- `_CONTROL_FLOW_GUARD` Defined as 1 when the `/guard:cf` (Enable Control Flow Guard) compiler option is set. Otherwise, undefined.

- `__COUNTER__` Expands to an integer literal that starts at 0. The value increments by 1 every time it's used in a source file, or in included headers of the source file. `__COUNTER__` remembers its state when you use precompiled headers. This macro is always defined.

  This example uses `__COUNTER__` to assign unique identifiers to three different objects of the same type. The `exampleClass` constructor takes an integer as a parameter. In `main`, the application declares three objects of type `exampleClass`, using `__COUNTER__` as the unique identifier parameter:

  C++

  ```
  // 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` Defined as the integer literal value 200406 when compiled as C++ and a /clr compiler option is set. Otherwise, undefined. When defined, `__cplusplus_cli` is in effect throughout the translation unit.

```
C++

// 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` Defined as the integer literal value 201009 when compiled as C++ and the `/ZW` (Windows Runtime Compilation) compiler option is set. Otherwise, undefined.

- `_CPPRTTI` Defined as 1 if the `/GR` (Enable Run-Time Type Information) compiler option is set. Otherwise, undefined.
- `_CPPUNWIND` Defined as 1 if one or more of the `/GX` (Enable Exception Handling), `/clr` (Common Language Runtime Compilation), or `/EH` (Exception Handling Model) compiler options are set. Otherwise, undefined.
- `_DEBUG` Defined as 1 when the /LDd, /MDd, or /MTd compiler option is set. Otherwise, undefined.
- `_DLL` Defined as 1 when the /MD or /MDd (Multithreaded DLL) compiler option is set. Otherwise, undefined.
- `__FUNCDNAME__` Defined as a string literal that contains the decorated name of the enclosing function. The macro is defined only within a function. The `__FUNCDNAME__` macro isn't expanded if you use the /EP or /P compiler option.

  This example uses the `__FUNCDNAME__`, `__FUNCSIG__`, and `__FUNCTION__` macros to display function information.

  C++

  ```
  // 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__` Defined as a string literal that contains the signature of the enclosing function. The macro is defined only within a function. The `__FUNCSIG__` macro isn't expanded if you use the /EP or /P compiler option. When compiled for a 64-bit target, the calling convention is `__cdecl` by default. For an example of usage, see the `__FUNCDNAME__` macro.
- `__FUNCTION__` Defined as a string literal that contains the undecorated name of the enclosing function. The macro is defined only within a function. The `__FUNCTION__`

macro isn't expanded if you use the /EP or /P compiler option. For an example of usage, see the `__FUNCDNAME__` macro.

- `_INTEGRAL_MAX_BITS` Defined as the integer literal value 64, the maximum size (in bits) for a non-vector integral type. This macro is always defined.

  C++

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

- `__INTELLISENSE__` Defined as 1 during an IntelliSense compiler pass in the Visual Studio IDE. Otherwise, undefined. You can use this macro to guard code the IntelliSense compiler doesn't understand, or use it to toggle between the build and IntelliSense compiler. For more information, see Troubleshooting Tips for IntelliSense Slowness.

- `_ISO_VOLATILE` Defined as 1 if the /volatile:iso compiler option is set. Otherwise, undefined.

- `_KERNEL_MODE` Defined as 1 if the `/kernel` (Create Kernel Mode Binary) compiler option is set. Otherwise, undefined.

- `_M_AMD64` Defined as the integer literal value 100 for compilations that target x64 processors or ARM64EC. Otherwise, undefined.

- `_M_ARM` Defined as the integer literal value 7 for compilations that target ARM processors. Undefined for ARM64, ARM64EC, and other targets.

- `_M_ARM_ARMV7VE` Defined as 1 when the /arch:ARMv7VE compiler option is set for compilations that target ARM processors. Otherwise, undefined.

- `_M_ARM_FP` Defined as an integer literal value that indicates which /arch compiler option was set for ARM processor targets. Otherwise, undefined.

  - A value in the range 30-39 if no `/arch` ARM option was specified, indicating the default architecture for ARM was set (`VFPv3`).

  - A value in the range 40-49 if `/arch:VFPv4` was set.

  - For more information, see `/arch` (ARM).

- `_M_ARM64` Defined as 1 for compilations that target ARM64. Otherwise, undefined.
- `_M_ARM64EC` Defined as 1 for compilations that target ARM64EC. Otherwise, undefined.
- `_M_CEE` Defined as 001 if any `/clr` (Common Language Runtime Compilation) compiler option is set. Otherwise, undefined.
- `_M_CEE_PURE` Deprecated beginning in Visual Studio 2015. Defined as 001 if the /clr:pure compiler option is set. Otherwise, undefined.
- `_M_CEE_SAFE` Deprecated beginning in Visual Studio 2015. Defined as 001 if the /clr:safe compiler option is set. Otherwise, undefined.
- `_M_FP_CONTRACT` Available beginning in Visual Studio 2022. Defined as 1 if the /fp:contract or /fp:fast compiler option is set. Otherwise, undefined.
- `_M_FP_EXCEPT` Defined as 1 if the /fp:except or /fp:strict compiler option is set. Otherwise, undefined.
- `_M_FP_FAST` Defined as 1 if the /fp:fast compiler option is set. Otherwise, undefined.
- `_M_FP_PRECISE` Defined as 1 if the /fp:precise compiler option is set. Otherwise, undefined.
- `_M_FP_STRICT` Defined as 1 if the /fp:strict compiler option is set. Otherwise, undefined.
- `_M_IX86` Defined as the integer literal value 600 for compilations that target x86 processors. This macro isn't defined for x64 or ARM compilation targets.
- `_M_IX86_FP` Defined as an integer literal value that indicates the /arch compiler option that was set, or the default. This macro is always defined when the compilation target is an x86 processor. Otherwise, undefined. When defined, the value is:
  - 0 if the `/arch:IA32` compiler option was set.
  - 1 if the `/arch:SSE` compiler option was set.
  - 2 if the `/arch:SSE2`, `/arch:AVX`, `/arch:AVX2`, or `/arch:AVX512` compiler option was set. This value is the default if an `/arch` compiler option wasn't specified. When `/arch:AVX` is specified, the macro `__AVX__` is also defined. When `/arch:AVX2` is specified, both `__AVX__` and `__AVX2__` are also defined. When

`/arch:AVX512` is specified, `__AVX__`, `__AVX2__`, `__AVX512BW__`, `__AVX512CD__`, `__AVX512DQ__`, `__AVX512F__`, and `__AVX512VL__` are also defined.

  - For more information, see `/arch` (x86).

- `_M_X64` Defined as the integer literal value 100 for compilations that target x64 processors or ARM64EC. Otherwise, undefined.

- `_MANAGED` Defined as 1 when the /clr compiler option is set. Otherwise, undefined.

- `_MSC_BUILD` Defined as an integer literal that contains the revision number element of the compiler's version number. The revision number is the last element of the period-delimited version number. For example, if the version number of the Microsoft C/C++ compiler is 15.00.20706.01, the `_MSC_BUILD` macro is 1. This macro is always defined.

- `_MSC_EXTENSIONS` Defined as 1 if the on-by-default `/Ze` (Enable Language Extensions) compiler option is set. Otherwise, undefined.

- `_MSC_FULL_VER` Defined as an integer literal that encodes the major, minor, and build number elements of the compiler's version number. The major number is the first element of the period-delimited version number, the minor number is the second element, and the build number is the third element.

  For example, if the Microsoft C/C++ compiler version is 19.39.33519, `_MSC_FULL_VER` is 193933519. Enter `cl /?` at the command line to view the compiler's version number. This macro is always defined. For more information about compiler versioning, see C++ compiler versioning and specifically Service releases starting with Visual Studio 2017 for more information about Visual Studio 2019 16.8, 16.9, 16.10 and 16.11, which require `_MSC_FULL_VER` to tell them apart.

- `_MSC_VER` Defined as an integer literal that encodes the major and minor number elements of the compiler's version number. The major number is the first element of the period-delimited version number and the minor number is the second element. For example, if the version number of the Microsoft C/C++ compiler is 17.00.51106.1, the value of `_MSC_VER` is 1700. Enter `cl /?` at the command line to view the compiler's version number. This macro is always defined.

  To test for compiler releases or updates in a given version of Visual Studio or later, use the `>=` operator. You can use it in a conditional directive to compare `_MSC_VER` against that known version. If you have several mutually exclusive versions to compare, order your comparisons in descending order of version number. For example, this code checks for compilers released in Visual Studio 2017 and later.

Next, it checks for compilers released in or after Visual Studio 2015. Then it checks for all compilers released before Visual Studio 2015:

```
C++
```

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

For more information about Visual Studio 2019 16.8 and 16.9, and 16.10 and 16.11, which share the same major and minor versions (and so have the same value for `_MSC_VER`), see Service releases starting with Visual Studio 2017.

For more information about the history of compiler versioning, and compiler version numbers and the Visual Studio versions they correspond to, see C++ compiler versioning. Also, Visual C++ Compiler Version on the Microsoft C++ team blog.

- `_MSVC_LANG` Defined as an integer literal that specifies the C++ language standard targeted by the compiler. Only code compiled as C++ sets it. The macro is the integer literal value `201402L` by default, or when the /std:c++14 compiler option is specified. The macro is set to `201703L` if the /std:c++17 compiler option is specified. The macro is set to `202002L` if the /std:c++20 compiler option is specified. It's set to a higher, unspecified value when the /std:c++latest option is specified. Otherwise, the macro is undefined. The `_MSVC_LANG` macro and /std (Specify language standard version) compiler options are available beginning in Visual Studio 2015 Update 3.

- `__MSVC_RUNTIME_CHECKS` Defined as 1 when one of the /RTC compiler options is set. Otherwise, undefined.

- `_MSVC_TRADITIONAL`:
  - Available beginning with Visual Studio 2017 version 15.8: Defined as 0 when the preprocessor conformance mode /experimental:preprocessor compiler option is set. Defined as 1 by default, or when the /experimental:preprocessor- compiler option is set, to indicate the traditional preprocessor is in use.
  - Available beginning with Visual Studio 2019 version 16.5: Defined as 0 when the preprocessor conformance mode /Zc:preprocessor compiler option is set. Defined as 1 by default, or when the `/Zc:preprocessor-` compiler option is set,

to indicate the traditional preprocessor is in use (essentially, `/Zc:preprocessor` replaces the deprecated `/experimental:preprocessor`).

```
C++

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

- `_MT` Defined as 1 when `/MD` or `/MDd` (Multithreaded DLL) or `/MT` or `/MTd` (Multithreaded) is specified. Otherwise, undefined.
- `_NATIVE_WCHAR_T_DEFINED` Defined as 1 when the /Zc:wchar_t compiler option is set. Otherwise, undefined.
- `_OPENMP` Defined as integer literal 200203, if the `/openmp` (Enable OpenMP 2.0 Support) compiler option is set. This value represents the date of the OpenMP specification implemented by MSVC. Otherwise, undefined.

```
C++

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

- `_PREFAST_` Defined as 1 when the /analyze compiler option is set. Otherwise, undefined.
- `__SANITIZE_ADDRESS__` Available beginning with Visual Studio 2019 version 16.9. Defined as 1 when the /fsanitize=address compiler option is set. Otherwise, undefined.
- `__TIMESTAMP__` Defined as a string literal that contains the date and time of the last modification of the current source file, in the abbreviated, constant length form returned by the CRT asctime function, for example, `Fri 19 Aug 13:32:58 2016`. This macro is always defined.
- `_VC_NODEFAULTLIB` Defined as 1 when the `/Zl` (Omit Default Library Name) compiler option is set. Otherwise, undefined.

- `_WCHAR_T_DEFINED` Defined as 1 when the default /Zc:wchar_t compiler option is set. The `_WCHAR_T_DEFINED` macro is defined but has no value if the `/Zc:wchar_t-` compiler option is set, and `wchar_t` is defined in a system header file included in your project. Otherwise, undefined.
- `_WIN32` Defined as 1 when the compilation target is 32-bit ARM, 64-bit ARM, x86, or x64. Otherwise, undefined.
- `_WIN64` Defined as 1 when the compilation target is 64-bit ARM or x64. Otherwise, undefined.
- `_WINRT_DLL` Defined as 1 when compiled as C++ and both `/ZW` (Windows Runtime Compilation) and `/LD` or `/LDd` compiler options are set. Otherwise, undefined.

No preprocessor macros that identify the ATL or MFC library version are predefined by the compiler. ATL and MFC library headers define these version macros internally. They're undefined in preprocessor directives made before the required header is included.

- `_ATL_VER` Defined in `<atldef.h>` as an integer literal that encodes the ATL version number.
- `_MFC_VER` Defined in `<afxver_.h>` as an integer literal that encodes the MFC version number.

See also

Macros (C/C++)
Preprocessor operators
Preprocessor directives

Feedback

Preprocessor grammar summary (C/C++)

Article • 08/03/2021

This article describes the formal grammar of the C and C++ preprocessor. It covers the syntax of preprocessing directives and operators. For more information, see Preprocessor and Pragma directives and the __pragma and _Pragma keywords.

Definitions for the grammar summary

Terminals are endpoints in a syntax definition. No other resolution is possible. Terminals include the set of reserved words and user-defined identifiers.

Nonterminals are placeholders in the syntax. Most are defined elsewhere in this syntax summary. Definitions can be recursive. The following nonterminals are defined in the Lexical conventions section of the *C++ Language Reference*:

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

An optional component is indicated by the subscripted $_{opt}$. For example, the following syntax indicates an optional expression enclosed in curly braces:

{ *expression*$_{opt}$ **}**

Document conventions

The conventions use different font attributes for different components of the syntax. The symbols and fonts are as follows:

Attribute	Description
nonterminal	Italic type indicates nonterminals.
#include	Terminals in bold type are literal reserved words and symbols that must be entered as shown. Characters in this context are always case sensitive.
opt	Nonterminals followed by $_{opt}$ are always optional.
default typeface	Characters in the set described or listed in this typeface can be used as terminals in statements.

A colon (`:`) following a nonterminal introduces its definition. Alternative definitions are listed on separate lines.

In code syntax blocks, these symbols in the default typeface have a special meaning:

Symbol	Description
[ ]	Square brackets surround an optional element.
{ | }	Curly braces surround alternative elements, separated by vertical bars.
...	Indicates the previous element pattern can be repeated.

In code syntax blocks, commas (`,`), periods (`.`), semi-colons (`;`), colons (`:`), parentheses (`( )`), double-quotes (`"`), and single-quotes (`'`) are literals.

Preprocessor grammar

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*

any other constant expression

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: one of

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

token-string:

String of *token*

token:

keyword

identifier

constant

operator

punctuator

filename:

Legal operating system filename

path-spec:

Legal file path

text:

Any sequence of text

> **Note**
>
> The following nonterminals are expanded in the **Lexical conventions** section of the *C++ Language Reference*: *constant*, *constant-expression*, *identifier*, *keyword*, *operator*, and *punctuator*.

See also

C/C++ preprocessor reference

Pragma directives and the `__pragma` and `_Pragma` keywords

Article • 03/31/2022

Pragma directives specify machine-specific or operating system-specific compiler features. A line that starts with `#pragma` specifies a pragma directive. The Microsoft-specific `__pragma` keyword enables you to code pragma directives within macro definitions. The standard `_Pragma` preprocessor operator, introduced in C99 and adopted by C++11, is similar.

Syntax

> **`#pragma`** *token-string*
> **`__pragma(`** *token-string* **`)`** // two leading underscores - Microsoft-specific extension
> **`_Pragma(`** *string-literal* **`)`** // C99

Remarks

Each implementation of C and C++ supports some features unique to its host machine or operating system. Some programs, for example, must exercise precise control over the location of data in memory, or control the way certain functions receive parameters. The **`#pragma`** directives offer a way for each compiler to offer machine- and operating system-specific features, while maintaining overall compatibility with the C and C++ languages.

Pragma directives are machine-specific or operating system-specific by definition, and are typically different for every compiler. A pragma can be used in a conditional directive, to provide new preprocessor functionality. Or, use one to provide implementation-defined information to the compiler.

The *token-string* is a series of characters representing a specific compiler instruction and arguments, if any. The number sign (`#`) must be the first non-white-space character on the line that contains the pragma. White-space characters can separate the number sign and the word "pragma". Following **`#pragma`**, write any text that the translator can parse as preprocessing tokens. The argument to **`#pragma`** is subject to macro expansion.

The *string-literal* is the input to `_Pragma`. Outer quotes and leading/trailing whitespace are removed. `\"` is replaced with `"` and `\\` is replaced with `\`.

The compiler issues a warning when it finds a pragma that it doesn't recognize, and continues compilation.

The Microsoft C and C++ compilers recognize the following pragma directives:

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] Supported only by the C++ compiler.

Pragma directives and compiler options

Some pragma directives provide the same functionality as compiler options. When a pragma is reached in source code, it overrides the behavior specified by the compiler option. For example, if you specified /Zp8, you can override this compiler setting for specific sections of the code with pack:

Windows Command Prompt

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

C++

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

The `__pragma` keyword

The compiler also supports the Microsoft-specific `__pragma` keyword, which has the same functionality as the `#pragma` directive. The difference is, the `__pragma` keyword is usable inline in a macro definition. The `#pragma` directive isn't usable in a macro definition, because the compiler interprets the number sign character ('#') in the directive as the stringizing operator (#).

The following code example demonstrates how the `__pragma` keyword can be used in a macro. This code is excerpted from the *mfcdual.h* header in the ACDUAL sample in

"Compiler COM Support Samples":

```
C++
```

```
#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; \
```

The `_Pragma` preprocessing operator

`_Pragma` is similar to the Microsoft-specific __pragma keyword. It was introduced into the C standard in C99, and the C++ standard in C++11. It's available in C only when you specify the /std:c11 or /std:c17 option. For C++, it's available in all `/std` modes, including the default.

Unlike `#pragma`, `_Pragma` allows you to put pragma directives into a macro definition. The string literal should be what you would otherwise put following a `#pragma` statement. For example:

```
C
```

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

Quotation marks and back-slashes should be escaped, as shown above. A pragma string that isn't recognized is ignored.

The following code example demonstrates how the `_Pragma` keyword could be used in an assert-like macro. It creates a pragma directive that suppresses a warning when the condition expression happens to be constant.

The macro definition uses the `do ... while(0)` idiom for multi-statement macros so that it can be used as though it were one statement. For more information, see C multi-

line macro on Stack Overflow. The `_Pragma` statement in the example only applies to the line of code that follows it.

C

```
// 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;
}
```

See also

C/C++ preprocessor reference

C pragma directives

Keywords

`alloc_text` pragma

Article • 08/03/2021

Names the code section where the specified function definitions are placed. The pragma must occur between a function declarator and the function definition for the named functions.

Syntax

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

Remarks

The **`alloc_text`** pragma doesn't handle C++ member functions or overloaded functions. It's applicable only to functions declared with C linkage, that is, functions declared with the **`extern "C"`** linkage specification. If you attempt to use this pragma on a function with C++ linkage, a compiler error is generated.

Since function addressing using **`__based`** isn't supported, specifying section locations requires the use of the **`alloc_text`** pragma. The name specified by *text-section* should be enclosed in double quotation marks.

The **`alloc_text`** pragma must appear after the declarations of any of the specified functions and before the definitions of these functions.

Functions referenced in an **`alloc_text`** pragma should be defined in the same module as the pragma. Otherwise, if an undefined function is later compiled into a different text section, the error may or may not be caught. Although the program will usually run correctly, the function won't be allocated in the intended sections.

Other limitations on **`alloc_text`** are as follows:

- It can't be used inside a function.
- It must be used after the function has been declared, but before the function has been defined.

See also

Pragma directives and the __pragma and _Pragma keywords

`auto_inline` pragma

Article • 08/03/2021

Excludes any functions defined within the range where `off` is specified from being considered as candidates for automatic inline expansion.

Syntax

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

Remarks

To use the `auto_inline` pragma, place it before and immediately after, not inside, a function definition. The pragma takes effect as soon as the first function definition after the pragma is seen.

See also

Pragma directives and the __pragma and _Pragma keywords

`bss_seg` pragma

Article • 08/03/2021

Specifies the section (segment) where uninitialized variables are stored in the object (.obj) file.

Syntax

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

Parameters

`push`
(Optional) Puts a record on the internal compiler stack. A **`push`** can have an *identifier* and *section-name*.

`pop`
(Optional) Removes a record from the top of the internal compiler stack. A **`pop`** can have an *identifier* and *section-name*. You can pop multiple records using just one **`pop`** command by using the *identifier*. The *section-name* becomes the active BSS section name after the pop.

identifier
(Optional) When used with **`push`**, assigns a name to the record on the internal compiler stack. When used with **`pop`**, the directive pops records off the internal stack until *identifier* is removed. If *identifier* isn't found on the internal stack, nothing is popped.

"*section-name*"
(Optional) The name of a section. When used with **`pop`**, the stack is popped and *section-name* becomes the active BSS section name.

"*section-class*"
(Optional) Ignored, but included for compatibility with versions of Microsoft C++ earlier than version 2.0.

Remarks

A *section* in an object file is a named block of data that's loaded into memory as a unit. A *BSS section* is a section that contains uninitialized data. In this article, the terms *segment* and *section* have the same meaning.

The **`bss_seg`** pragma directive tells the compiler to put all uninitialized data items from the translation unit into a BSS section named *section-name*. In some cases, use of **`bss_seg`** can speed load times by grouping uninitialized data into one section. By default, the BSS section used for uninitialized data in an object file is named `.bss`. A **`bss_seg`** pragma directive without a *section-name* parameter resets the BSS section name for the subsequent uninitialized data items to `.bss`.

Data allocated using the **`bss_seg`** pragma does not retain any information about its location.

For a list of names that shouldn't be used to create a section, see /SECTION.

You can also specify sections for initialized data (data_seg), functions (code_seg), and const variables (const_seg).

You can use the DUMPBIN.EXE application to view object files. Versions of DUMPBIN for each supported target architecture are included with Visual Studio.

Example

C++

```
// 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() {
}
```

See also

Pragma directives and the __pragma and _Pragma keywords

`check_stack` pragma

Article • 08/03/2021

Instructs the compiler to turn off stack probes if **`off`** (or **`-`**) is specified, or to turn on stack probes if **`on`** (or **`+`**) is specified.

Syntax

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

Remarks

This pragma takes effect at the first function defined after the pragma is seen. Stack probes are neither a part of macros nor of functions that are generated inline.

If you don't give an argument for the **`check_stack`** pragma, stack checking reverts to the behavior specified on the command line. For more information, see Compiler options. The interaction of the `#pragma check_stack` and the /Gs option is summarized in the following table.

Using the check_stack Pragma

Syntax	**Compiled with `/Gs` option?**	**Action**
`#pragma check_stack( )` or `#pragma check_stack`	Yes	Turns off stack checking for functions that follow
`#pragma check_stack( )` or `#pragma check_stack`	No	Turns on stack checking for functions that follow
`#pragma check_stack(on)` or `#pragma check_stack +`	Yes or No	Turns on stack checking for functions that follow

Syntax	Compiled with `/Gs` option?	Action
`#pragma check_stack(off)` or `#pragma check_stack -`	Yes or No	Turns off stack checking for functions that follow

See also

Pragma directives and the __pragma and _Pragma keywords

`code_seg` pragma

Article • 08/03/2021

Specifies the text section (segment) where functions are stored in the object (.obj) file.

Syntax

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

Parameters

`push`
(Optional) Puts a record on the internal compiler stack. A **`push`** can have an *identifier* and *section-name*.

`pop`
(Optional) Removes a record from the top of the internal compiler stack. A **`pop`** can have an *identifier* and *section-name*. You can pop multiple records using just one **`pop`** command by using the *identifier*. The *section-name* becomes the active text section name after the pop.

identifier
(Optional) When used with **`push`**, assigns a name to the record on the internal compiler stack. When used with **`pop`**, the directive pops records off the internal stack until *identifier* is removed. If *identifier* isn't found on the internal stack, nothing is popped.

"*section-name*"
(Optional) The name of a section. When used with **`pop`**, the stack is popped and *section-name* becomes the active text section name.

"*section-class*"
(Optional) Ignored, but included for compatibility with versions of Microsoft C++ earlier than version 2.0.

Remarks

A *section* in an object file is a named block of data that's loaded into memory as a unit. A *text section* is a section that contains executable code. In this article, the terms *segment* and *section* have the same meaning.

The `code_seg` pragma directive tells the compiler to put all subsequent object code from the translation unit into a text section named *section-name*. By default, the text section used for functions in an object file is named `.text`. A `code_seg` pragma directive without a *section-name* parameter resets the text section name for the subsequent object code to `.text`.

The `code_seg` pragma directive doesn't control placement of object code generated for instantiated templates. Nor does it control code generated implicitly by the compiler, such as special member functions. To control that code, we recommend you use the __declspec(code_seg(...)) attribute instead. It gives you control over placement of all object code, including compiler-generated code.

For a list of names that shouldn't be used to create a section, see /SECTION.

You can also specify sections for initialized data (data_seg), uninitialized data (bss_seg), and const variables (const_seg).

You can use the DUMPBIN.EXE application to view object files. Versions of DUMPBIN for each supported target architecture are included with Visual Studio.

Example

This example shows how to use the **code_seg** pragma directive to control where object code is put:

C++

```
// 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() {
}
```

See also

code_seg (__declspec)

Pragma directives and the __pragma and _Pragma keywords

`comment` pragma

Article • 08/03/2021

Places a comment record into an object file or executable file.

Syntax

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

Remarks

The *comment-type* is one of the predefined identifiers, described below, that specifies the type of comment record. The optional *comment-string* is a string literal that provides additional information for some comment types. Because *comment-string* is a string literal, it obeys all the rules for string literals on use of escape characters, embedded quotation marks (`"`), and concatenation.

compiler

Places the name and version number of the compiler in the object file. This comment record is ignored by the linker. If you supply a *comment-string* parameter for this record type, the compiler generates a warning.

lib

Places a library-search record in the object file. This comment type must be accompanied by a *comment-string* parameter that has the name (and possibly the path) of the library that you want the linker to search. The library name follows the default library-search records in the object file. The linker searches for this library the same way as if you specified it on the command line, as long as the library isn't specified by using /nodefaultlib. You can place multiple library-search records in the same source file. Each record appears in the object file in the same order it's found in the source file.

If the order of the default library and an added library is important, compiling with the /Zl switch will prevent the default library name from being placed in the object module. A second comment pragma then can be used to insert the name of the default library after the added library. The libraries listed with these pragma directives will appear in the object module in the same order they're found in the source code.

linker

Places a linker option in the object file. You can use this comment-type to specify a linker option instead of passing it to the command line or specifying it in the development environment. For example, you can specify the /include option to force the inclusion of a symbol:

```
C

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

Only the following (*comment-type*) linker options are available to be passed to the linker identifier:

- /DEFAULTLIB
- /EXPORT
- /INCLUDE
- /MANIFESTDEPENDENCY
- /MERGE
- /SECTION

user

Places a general comment in the object file. The *comment-string* parameter contains the text of the comment. This comment record is ignored by the linker.

Examples

The following pragma causes the linker to search for the EMAPI.LIB library while linking. The linker searches first in the current working directory, and then in the path specified in the LIB environment variable.

```
C

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

The following pragma causes the compiler to place the name and version number of the compiler in the object file:

C

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

For comments that take a *comment-string* parameter, you can use a macro in any place where you would use a string literal, as long as the macro expands to a string literal. You can also concatenate any combination of string literals and macros that expand to string literals. For example, the following statement is acceptable:

C

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

See also

Pragma directives and the __pragma and _Pragma keywords

`component` pragma

Article • 08/03/2021

Controls the collection of browse information or dependency information from within source files.

Syntax

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

Remarks

Browser

You can turn collecting on or off, and you can specify particular names to be ignored as information is collected.

Using on or off controls the collection of browse information from the pragma forward. For example:

```
C++

#pragma component(browser, off)
```

stops the compiler from collecting browse information.

> ⓘ **Note**
>
> To turn on the collecting of browse information with this pragma, **browse information must first be enabled**.

The `references` option can be used with or without the *name* argument. Using `references` without *name* turns on or off the collecting of references (other browse information continues to be collected, however). For example:

```
C++
```

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

stops the compiler from collecting reference information.

Using **`references`** with *name* and **`off`** prevents references to *name* from appearing in the browse information window. Use this syntax to ignore names and types you are not interested in and to reduce the size of browse information files. For example:

```
C++

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

ignores references to DWORD from that point forward. You can turn collecting of references to DWORD back on by using **`on`**:

```
C++

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

This is the only way to resume collecting references to *name*; you must explicitly turn on any *name* that you have turned off.

To prevent the preprocessor from expanding *name* (such as expanding NULL to 0), put quotes around it:

```
C++

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

Minimal rebuild

The deprecated /Gm (Enable Minimal Rebuild) feature requires the compiler to create and store C++ class dependency information, which takes disk space. To save disk space, you can use `#pragma component( minrebuild, off )` whenever you don't need to collect dependency information, for instance, in unchanging header files. Insert `#pragma component( minrebuild, on )` after unchanging classes to turn dependency collection back on.

Reduce type information

The `mintypeinfo` option reduces the debugging information for the region specified. The volume of this information is considerable, impacting .pdb and .obj files. You cannot debug classes and structures in the `mintypeinfo` region. Use of the `mintypeinfo` option can be helpful to avoid the following warning:

Windows Command Prompt

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

For more information, see the /Gm (Enable Minimal Rebuild) compiler option.

See also

Pragma directives and the __pragma and _Pragma keywords

`conform` pragma

Article • 08/03/2021

C++ Specific

Specifies the run-time behavior of the /Zc:forScope compiler option.

Syntax

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

Parameters

name
Specifies the name of the compiler option to be modified. The only valid *name* is `forScope`.

`show`
(Optional) Causes the current setting of *name* (true or false) to be displayed by means of a warning message during compilation. For example, `#pragma conform(forScope, show)`.

`on`, **`off`**
(Optional) Setting *name* to **`on`** enables the /Zc:forScope compiler option. The default is **`off`**.

`push`
(Optional) Pushes the current value of *name* onto the internal compiler stack. If you specify *identifier*, you can specify the **`on`** or **`off`** value for *name* to be pushed onto the stack. For example, `#pragma conform(forScope, push, myname, on)`.

`pop`
(Optional) Sets the value of *name* to the value at the top of the internal compiler stack and then pops the stack. If identifier is specified with **`pop`**, the stack will be popped back until it finds the record with *identifier*, which will also be popped; the current value for *name* in the next record on the stack becomes the new value for *name*. If you specify **`pop`** with an *identifier* that is not in a record on the stack, the **`pop`** is ignored.

identifier
(Optional) Can be included with a **`push`** or **`pop`** command. If *identifier* is used, then an

`on` or `off` specifier can also be used.

Example

C++

```
// 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() {}
```

See also

Pragma directives and the __pragma and _Pragma keywords

`const_seg` pragma

Article • 08/03/2021

Specifies the section (segment) where const variables are stored in the object (.obj) file.

Syntax

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

Parameters

`push`
(Optional) Puts a record on the internal compiler stack. A **`push`** can have an *identifier* and *section-name*.

`pop`
(Optional) Removes a record from the top of the internal compiler stack. A **`pop`** can have an *identifier* and *section-name*. You can pop multiple records using just one **`pop`** command by using the *identifier*. The *section-name* becomes the active const section name after the pop.

identifier
(Optional) When used with **`push`**, assigns a name to the record on the internal compiler stack. When used with **`pop`**, the directive pops records off the internal stack until *identifier* is removed. If *identifier* isn't found on the internal stack, nothing is popped.

"*section-name*"
(Optional) The name of a section. When used with **`pop`**, the stack is popped and *section-name* becomes the active const section name.

"*section-class*"
(Optional) Ignored, but included for compatibility with versions of Microsoft C++ earlier than version 2.0.

Remarks

A *section* in an object file is a named block of data that's loaded into memory as a unit. A *const section* is a section that contains constant data. In this article, the terms *segment* and *section* have the same meaning.

The **`const_seg`** pragma directive tells the compiler to put all constant data items from the translation unit into a const section named *section-name*. The default section in the object file for **`const`** variables is `.rdata`. Some **`const`** variables, such as scalars, are automatically inlined into the code stream. Inlined code doesn't appear in `.rdata`. A **`const_seg`** pragma directive without a *section-name* parameter resets the section name for the subsequent **`const`** data items to `.rdata`.

If you define an object that requires dynamic initialization in a `const_seg`, the result is undefined behavior.

For a list of names that shouldn't be used to create a section, see /SECTION.

You can also specify sections for initialized data (data_seg), uninitialized data (bss_seg), and functions (code_seg).

You can use the DUMPBIN.EXE application to view object files. Versions of DUMPBIN for each supported target architecture are included with Visual Studio.

Example

C++

```
// 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;
}
```

Output

```
test1
test2
test3
test4
```

See also

Pragma directives and the __pragma and _Pragma keywords

`data_seg` pragma

Article • 08/03/2021

Specifies the data section (segment) where initialized variables are stored in the object (.obj) file.

Syntax

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

Parameters

`push`
(Optional) Puts a record on the internal compiler stack. A **`push`** can have an *identifier* and *section-name*.

`pop`
(Optional) Removes a record from the top of the internal compiler stack. A **`pop`** can have an *identifier* and *section-name*. You can pop multiple records using just one **`pop`** command by using the *identifier*. The *section-name* becomes the active data section name after the pop.

identifier
(Optional) When used with **`push`**, assigns a name to the record on the internal compiler stack. When used with **`pop`**, pops records off the internal stack until *identifier* is removed. If *identifier* isn't found on the internal stack, nothing is popped.

identifier enables multiple records to be popped with a single **`pop`** command.

"*section-name*"
(Optional) The name of a section. When used with **`pop`**, the stack is popped and *section-name* becomes the active data section name.

"*section-class*"
(Optional) Ignored, but included for compatibility with versions of Microsoft C++ earlier than version 2.0.

Remarks

A *section* in an object file is a named block of data that's loaded into memory as a unit. A *data section* is a section that contains initialized data. In this article, the terms *segment* and *section* have the same meaning.

The default section in the .obj file for initialized variables is `.data`. Variables that are uninitialized are considered to be initialized to zero and are stored in `.bss`.

The **`data_seg`** pragma directive tells the compiler to put all initialized data items from the translation unit into a data section named *section-name*. By default, the data section used for initialized data in an object file is named `.data`. Variables that are uninitialized are considered to be initialized to zero, and are stored in `.bss`. A **`data_seg`** pragma directive without a *section-name* parameter resets the data section name for the subsequent initialized data items to `.data`.

Data allocated using **`data_seg`** doesn't retain any information about its location.

For a list of names that shouldn't be used to create a section, see /SECTION.

You can also specify sections for const variables (const_seg), uninitialized data (bss_seg), and functions (code_seg).

You can use the DUMPBIN.EXE application to view object files. Versions of DUMPBIN for each supported target architecture are included with Visual Studio.

Example

C++

```
// 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() {
}
```

See also

Pragma directives and the __pragma and _Pragma keywords

`deprecated` pragma

Article • 08/03/2021

The `deprecated` pragma lets you indicate that a function, type, or any other identifier may no longer be supported in a future release or should no longer be used.

> **Note**
>
> For information about the C++14 `[[deprecated]]` attribute, and guidance on when to use that attribute instead of the Microsoft `__declspec(deprecated)` modifier or the `deprecated` pragma, see **Attributes in C++**.

Syntax

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

Remarks

When the compiler encounters an identifier specified by a `deprecated` pragma, it issues compiler warning C4995.

You can deprecate macro names. Place the macro name in quotes or else macro expansion will occur.

Because the `deprecated` pragma works on all matching identifiers, and does not take signatures into account, it is not the best option for deprecating specific versions of overloaded functions. Any matching function name that is brought into scope triggers the warning.

We recommend you use the C++14 `[[deprecated]]` attribute, when possible, instead of the `deprecated` pragma. The Microsoft-specific __declspec(deprecated) declaration modifier is also a better choice in many cases than the `deprecated` pragma. The `[[deprecated]]` attribute and `__declspec(deprecated)` modifier allow you to specify deprecated status for particular forms of overloaded functions. The diagnostic warning only appears on references to the specific overloaded function the attribute or modifier applies to.

Example

C++

```
// 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
}
```

The following sample shows how to deprecate a class:

C++

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

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

See also

Pragma directives and the __pragma and _Pragma keywords

`detect_mismatch` pragma

Article • 08/03/2021

Places a record in an object. The linker checks these records for potential mismatches.

Syntax

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

Remarks

When you link the project, the linker throws a LNK2038 error if the project contains two objects that have the same *name* but each has a different *value*. Use this pragma to prevent inconsistent object files from linking.

Both *name* and *value* are string literals and obey the rules for string literals with respect to escape characters and concatenation. They are case-sensitive and cannot contain a comma, equal sign, quotation marks, or the **null** character.

Example

This example creates two files that have different version numbers for the same version label.

C++

```
// 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")
```

If you compile both of these files by using the command line `cl pragma_directive_detect_mismatch_a.cpp pragma_directive_detect_mismatch_b.cpp`, you will receive the error LNK2038.

See also

Pragma directives and the __pragma and _Pragma keywords

`execution_character_set` pragma

Article • 02/01/2022

Specifies the execution character set used for string and character literals. This directive isn't needed for literals marked with the `u8` prefix.

Syntax

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

Parameters

`target`
Specifies the target execution character set. Currently the only target execution set supported is `"utf-8"`.

Remarks

This compiler directive is obsolete in Visual Studio 2015 Update 2 and later versions. We recommend that you use the **`/execution-charset:utf-8`** or **`/utf-8`** compiler options together with the `u8` prefix on narrow character and string literals that contain extended characters. For more information about the `u8` prefix, see String and character literals. For more information about the compiler options, see /execution-charset (Set execution character set) and /utf-8 (Set source and execution character sets to UTF-8).

The `#pragma execution_character_set("utf-8")` directive tells the compiler to encode narrow character and narrow string literals in your source code as UTF-8 in the executable. This output encoding is independent of how the source file is encoded.

By default, when the compiler encodes narrow characters and narrow strings, it uses the current code page as the execution character set. Unicode or DBCS characters outside the range of the current code page get converted to the default replacement character in the output. Unicode and DBCS characters are truncated to their low-order byte, which is almost never what you intend. To specify the UTF-8 encoding for literals in the source file, use a `u8` prefix. The compiler passes these UTF-8 encoded strings to the output unchanged. Narrow character literals prefixed by `u8` must fit in a byte, or they're truncated on output.

By default, Visual Studio uses the current code page as the source character set used to interpret your source code for output. When a file is read in, Visual Studio interprets it according to the current code page unless the file has a code page set. Or, unless a byte-order mark (BOM) or UTF-16 characters are detected at the beginning of the file. You can't set UTF-8 as the current code page in some versions of Windows. When the automatic detection finds UTF-8 encoded source files without a BOM in those versions, Visual Studio assumes they're encoded in the current code page. Characters in the source file that are outside the range of the specified or automatically detected code page can cause compiler warnings and errors.

See also

Pragma directives and the __pragma and _Pragma keywords
/execution-charset (Set execution character set)
/utf-8 (Set source and execution character sets to UTF-8)

`fenv_access` pragma

Article • 10/25/2021

Disables (`on`) or enables (`off`) optimizations that could change floating-point environment flag tests and mode changes.

Syntax

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

Remarks

By default, `fenv_access` is `off`. The compiler assumes your code doesn't access or manipulate the floating-point environment. If environment access isn't required, the compiler can do more to optimize your floating-point code.

Enable `fenv_access` if your code tests floating-point status flags, exceptions, or sets control mode flags. The compiler disables floating-point optimizations, so your code can access the floating-point environment consistently.

The /fp:strict command-line option automatically enables `fenv_access`. For more information on this and other floating-point behavior, see /fp (Specify Floating-Point Behavior).

There are restrictions on the ways you can use the `fenv_access` pragma in combination with other floating-point settings:

- You can't enable `fenv_access` unless precise semantics are enabled. Precise semantics can be enabled either by the float_control pragma, or by using the /fp:precise or /fp:strict compiler options. The compiler defaults to `/fp:precise` if no other floating-point command-line option is specified.
- You can't use `float_control` to disable precise semantics when `fenv_access(on)` is set.

The `fenv_access(on)` directive disables generation of floating-point *contractions*, machine instructions that combine floating-point operations. `fenv_access(off)` restores the previous behavior for contractions. This behavior is new in Visual Studio 2022. Previous compiler versions could generate contractions by default under

`fenv_access(on)`. For more information about floating-point contractions, see /fp:contract.

The kinds of optimizations that are subject to `fenv_access` are:

- Global common subexpression elimination
- Code motion
- Constant folding

Other floating-point pragma directives include:

- float_control
- fp_contract

Examples

This example sets `fenv_access` to `on` to set the floating-point control register for 24-bit precision:

C++

```
// 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);
}
```

Output

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

If you comment out `#pragma fenv_access (on)` from the previous sample, the output is different. It's because the compiler does compile-time evaluation, which doesn't use the control mode.

C++

```cpp
// 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);
}
```

Output

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

See also

Pragma directives and the __pragma and _Pragma keywords

`float_control` pragma

Article • 10/25/2021

Specifies floating-point behavior for a function.

Syntax

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

Options

`precise`, **`on`** | **`off`**, **`push`**
Specifies whether to enable (**`on`**) or disable (**`off`**) precise floating-point semantics. For information on differences with the **`/fp:precise`** compiler option, see the Remarks section. The optional **`push`** token pushes the current setting for **`float_control`** on the internal compiler stack.

`except`, **`on`** | **`off`**, **`push`**
Specifies whether to enable (**`on`**) or disable (**`off`**) floating-point exception semantics. The optional **`push`** token pushes the current setting for **`float_control`** on the internal compiler stack.

`except` can only be set to **`on`** when **`precise`** is also set to **`on`**.

`push`
Pushes the current **`float_control`** setting on to the internal compiler stack.

`pop`
Removes the **`float_control`** setting from the top of the internal compiler stack, and makes that the new **`float_control`** setting.

Remarks

The **`float_control`** pragma doesn't have the same behavior as the /fp compiler option. The **`float_control`** pragma only governs part of the floating-point behavior. It must be

combined with fp_contract and fenv_access pragma directives to recreate the `/fp` compiler options. The following table shows the equivalent pragma settings for each compiler option:

Option	`float_control(precise, *)`	`float_control(except, *)`	`fp_contract(*)`	`fenv_access(*)`
`/fp:strict`	`on`	`on`	`off`	`on`
`/fp:precise`	`on`	`off`	`off`*	`off`
`/fp:fast`	`off`	`off`	`on`	`off`

* In versions of Visual Studio before Visual Studio 2022, the `/fp:precise` behavior defaulted to `fp_contract(on)`.

In other words, you may need to use several pragma directives in combination to emulate the `/fp:fast`, `/fp:precise`, and `/fp:strict` command-line options.

There are restrictions on the ways you can use the `float_control` and `fenv_access` floating-point pragma directives in combination:

- You can only use `float_control` to set `except` to `on` if precise semantics are enabled. Precise semantics can be enabled either by the `float_control` pragma, or by using the `/fp:precise` or `/fp:strict` compiler options.
- You can't use `float_control` to turn off `precise` when exception semantics are enabled, whether by a `float_control` pragma or a `/fp:except` compiler option.
- You can't enable `fenv_access` unless precise semantics are enabled, whether by a `float_control` pragma or a compiler option.
- You can't use `float_control` to turn off `precise` when `fenv_access` is enabled.

These restrictions mean the order of some floating-point pragma directives is significant. To go from a fast model to a strict model using pragma directives, use the following code:

C++

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

To go from a strict model to a fast model by using the `float_control` pragma, use the following code:

```
C++
```

```
#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
```

If no options are specified, `float_control` has no effect.

The `float_control` directive disables contractions when it turns on `precise` or `except`. Use of `float_control` to turn off `precise` or `except` restores the previous setting for contractions. You can use the `fp_contract` pragma directive to change the compiler behavior on contractions. `float_control(push)` and `float_control(pop)` push and pop the setting for contractions as part of the `float_control` setting on to the internal compiler stack. This behavior is new in Visual Studio 2022. The `float_control` directive in previous compiler versions did not affect contraction settings.

Example

The following sample shows how to catch an overflow floating-point exception by using pragma `float_control`.

```
C++
```

```
// 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);
   }
}
```

Output

```
Pass
```

See also

Pragma directives and the __pragma and _Pragma keywords

fenv_access pragma

fp_contract pragma

`fp_contract` pragma

Article • 03/31/2022

Determines whether floating-point contraction takes place. A floating-point contraction is an instruction such as Fused-Multiply-Add (FMA) that combines two separate floating point operations into a single instruction. Use of these instructions can affect floating-point precision, because instead of rounding after each operation, the processor may round only once after both operations.

Syntax

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

Remarks

When you use the default compiler options, **`fp_contract`** is **`off`**, which tells the compiler to preserve individual floating-point instructions. Set **`fp_contract`** to **`on`** to use floating-point contraction instructions where possible. This behavior is new in Visual Studio 2022 version 17.0. In previous compiler versions, **`fp_contract`** defaulted to **`on`**.

For more information on floating-point behavior, see /fp (Specify floating-point behavior).

Other floating-point pragma directives include:

- fenv_access
- float_control

Example

The **`/fp:fast`** compiler option enables contractions by default, but the `#pragma fp_contract (off)` directive in this example turns them off. The code generated from this sample won't use a fused-multiply-add instruction even when it's available on the target processor. If you comment out `#pragma fp_contract (off)`, the generated code may use a fused-multiply-add instruction if it's available.

```
C++
```

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

#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);
   }
}
```

Output

```
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
```

See also

Pragma directives and the __pragma and _Pragma keywords

`function` pragma

Article • 08/03/2021

Tells the compiler to generate calls to functions specified in the pragma's argument list, instead of inlining them.

Syntax

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

Remarks

Intrinsic functions are normally generated as inline code, not as function calls. If you use the intrinsic pragma or the /Oi compiler option to tell the compiler to generate intrinsic functions, you can use the **`function`** pragma to explicitly force a function call. Once a **`function`** pragma is seen, it takes effect at the first function definition that contains a specified intrinsic function. The effect continues to the end of the source file, or to the appearance of an `intrinsic` pragma specifying the same intrinsic function. You can only use the **`function`** pragma outside of a function, at the global level.

For lists of the functions that have intrinsic forms, see intrinsic pragma.

Example

C++

```
// 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, '!'));
}
```

Output

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

See also

Pragma directives and the __pragma and _Pragma keywords

`hdrstop` pragma

Article • 08/03/2021

Gives you more control over precompilation file names, and over the location at which the compilation state is saved.

Syntax

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

Remarks

The *filename* is the name of the precompiled header file to use or create (depending on whether /Yu or /Yc is specified). If *filename* doesn't contain a path specification, the precompiled header file is assumed to be in the same directory as the source file.

If a C or C++ file contains a **`hdrstop`** pragma when compiled with **`/Yc`**, the compiler saves the state of the compilation up to the location of the pragma. The compiled state of any code that follows the pragma isn't saved.

Use *filename* to name the precompiled header file in which the compiled state is saved. A space between **`hdrstop`** and *filename* is optional. The file name specified in the **`hdrstop`** pragma is a string, and is subject to the constraints of any C or C++ string. In particular, you must enclose it in quotation marks and use the escape character (backslash, **`\`**) to specify directory names. For example:

```
C

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

The name of the precompiled header file is determined according to the following rules, in order of precedence:

1. The argument to the **`/Fp`** compiler option
2. The *filename* argument to `#pragma hdrstop`
3. The base name of the source file with a PCH extension

If none of the **`/Yc`** and **`/Yu`** options or the **`hdrstop`** pragma specifies a file name, the base name of the source file is used as the base name of the precompiled header file.

You can also use preprocessing commands to perform macro replacement as follows:

```
C

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

The following rules govern where the `hdrstop` pragma can be placed:

- It must appear outside any data or function declaration or definition.
- It must be specified in the source file, not within a header file.

Example

```
C

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

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

In this example, the `hdrstop` pragma appears after two files have been included and an inline function has been defined. This location might, at first, seem to be an odd placement for the pragma. Consider, however, that using the manual precompilation options, `/Yc` and `/Yu`, with the `hdrstop` pragma makes it possible for you to precompile entire source files, or even inline code. The Microsoft compiler doesn't limit you to precompiling only data declarations.

See also

Pragma directives and the __pragma and _Pragma keywords

`include_alias` pragma

Article • 08/03/2021

Specifies that when *alias_filename* is found in a `#include` directive, the compiler substitutes *actual_filename* in its place.

Syntax

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

Remarks

The **`include_alias`** pragma directive allows you to substitute files that have different names or paths for the file names included by source files. For example, some file systems allow longer header filenames than the 8.3 FAT file system limit. The compiler can't just truncate the longer names to 8.3, because the first eight characters of the longer header filenames may not be unique. Whenever the compiler sees the *alias_filename* string in a `#include` directive, it substitutes the name *actual_filename* instead. Then it loads the *actual_filename* header file. This pragma must appear before the corresponding `#include` directives. For example:

C++

```
// 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"
```

The alias to search for must match the specification exactly. The case, spelling, and the use of double quotation marks or angle brackets must all match. The **`include_alias`** pragma does simple string matching on the filenames. No other filename validation is performed. For example, given the following directives,

C++

```
#pragma include_alias("mymath.h", "math.h")
#include "./mymath.h"
#include "sys/mymath.h"
```

no alias substitution is done, since the header file strings don't match exactly. Also, header filenames used as arguments to the `/Yu` and `/Yc` compiler options, or the `hdrstop` pragma, aren't substituted. For example, if your source file contains the following directive,

C++

```
#include <AppleSystemHeaderStop.h>
```

the corresponding compiler option should be

> `/YcAppleSystemHeaderStop.h`

You can use the `include_alias` pragma to map any header filename to another. For example:

C++

```
#pragma include_alias( "api.h", "c:\version1.0\api.h" )
#pragma include_alias( <stdio.h>, <newstdio.h> )
#include "api.h"
#include <stdio.h>
```

Don't mix filenames enclosed in double quotation marks with filenames enclosed in angle brackets. For example, given the above two `#pragma include_alias` directives, the compiler does no substitution on the following `#include` directives:

C++

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

Furthermore, the following directive generates an error:

C++

```
#pragma include_alias(<header.h>, "header.h")  // Error
```

The filename reported in error messages, or as the value of the predefined `__FILE__` macro, is the name of the file after the substitution is done. For example, see the output after the following directives:

C++

```
#pragma include_alias( "VERYLONGFILENAME.H", "myfile.h" )
#include "VERYLONGFILENAME.H"
```

An error in *`VERYLONGFILENAME.H`* produces the following error message:

Output

```
myfile.h(15) : error C2059 : syntax error
```

Also note that transitivity isn't supported. Given the following directives,

C++

```
#pragma include_alias( "one.h", "two.h" )
#pragma include_alias( "two.h", "three.h" )
#include "one.h"
```

the compiler searches for the file *`two.h`* rather than *`three.h`*.

See also

Pragma directives and the __pragma and _Pragma keywords

`init_seg` pragma

Article • 08/03/2021

C++ Specific

Specifies a keyword or code section that affects the order in which startup code is executed.

Syntax

> **`#pragma init_seg(`** { **`compiler`** | **`lib`** | **`user`** | "*section-name*" [**`,`** *func-name*] } **`)`**

Remarks

The terms *segment* and *section* have the same meaning in this article.

Because code is sometimes required to initialize global static objects, you must specify when to construct the objects. In particular, it's important to use the **`init_seg`** pragma in dynamic-link libraries (DLLs), or in libraries that require initialization.

The options to the **`init_seg`** pragma are:

`compiler`
Reserved for Microsoft C run-time library initialization. Objects in this group are constructed first.

`lib`
Available for third-party class-library vendors' initializations. Objects in this group are constructed after the ones marked as **`compiler`**, but before any others.

`user`
Available to any user. Objects in this group are constructed last.

section-name
Allows explicit specification of the initialization section. Objects in a user-specified *section-name* aren't implicitly constructed. However, their addresses are placed in the section named by *section-name*.

The *section-name* you give will contain pointers to helper functions that will construct the global objects declared after the pragma in that module.

For a list of names you shouldn't use when creating a section, see /SECTION.

func-name
Specifies a function to be called in place of `atexit` when the program exits. This helper function also calls atexit with a pointer to the destructor for the global object. If you specify a function identifier in the pragma of the form,

```
C++

int __cdecl myexit (void (__cdecl *pf)(void))
```

then your function will be called instead of the C run-time library's `atexit`. It allows you to build a list of the destructors to call when you're ready to destroy the objects.

If you need to defer initialization (for example, in a DLL) you may choose to specify the section name explicitly. Your code must then call the constructors for each static object.

There are no quotes around the identifier for the `atexit` replacement.

Your objects will still be placed in the sections defined by the other `XXX_seg` pragma directives.

The objects that are declared in the module aren't automatically initialized by the C run-time. Your code has to do the initialization.

By default, `init_seg` sections are read only. If the section name is `.CRT`, the compiler silently changes the attribute to read only, even if it's marked as read, write.

You can't specify **`init_seg`** more than once in a translation unit.

Even if your object doesn't have a user-defined constructor, one explicitly defined in code, the compiler may generate one for you. For example, it may create one to bind v-table pointers. When needed, your code calls the compiler-generated constructor.

Example

```
C++

// 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();
}
```

Output

```
A()
A()
A()
~A()
~A()
~A()
```

See also

Pragma directives and the __pragma and _Pragma keywords

`inline_depth` pragma

Article • 08/03/2021

Specifies the inline heuristic search depth. Functions at a depth in the call graph greater than the specified value aren't inlined.

Syntax

> `#pragma inline_depth(` [*n*] `)`

Remarks

This pragma controls the inlining of functions marked inline and __inline, or inlined automatically under the `/Ob` compiler option. For more information, see /Ob (Inline function expansion).

n can be a value between 0 and 255, where 255 means unlimited depth in the call graph. A value of 0 inhibits inline expansion. When *n* isn't specified, the default value 254 is used.

The `inline_depth` pragma controls the number of times a series of function calls can be expanded. For example, assume the inline depth is 4. If A calls B, and B then calls C, all three calls are expanded inline. However, if the closest inline depth expansion is 2, only A and B are expanded, and C remains as a function call.

To use this pragma, you must set the `/Ob` compiler option to 1 or higher. The depth set using this pragma takes effect at the first function call after the pragma.

The inline depth can be decreased during expansion, but not increased. If the inline depth is 6, and during expansion the preprocessor encounters an `inline_depth` pragma with a value of 8, the depth remains 6.

The `inline_depth` pragma has no effect on functions marked with `__forceinline`.

> **Note**
>
> Recursive functions can be substituted inline to a maximum depth of 16 calls.

See also

`inline_recursion` pragma

Article • 08/03/2021

Controls the inline expansion of direct or mutually recursive function calls.

Syntax

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

Remarks

Use this pragma to control functions marked as inline and __inline or functions that the compiler automatically expands under the `/Ob2` option. Use of this pragma requires an /Ob compiler option setting of either 1 or 2. The default state for `inline_recursion` is off. This pragma takes effect at the first function call after the pragma is seen and doesn't affect the definition of the function.

The `inline_recursion` pragma controls how recursive functions are expanded. If `inline_recursion` is off, and if an inline function calls itself, either directly or indirectly, the function is expanded only one time. If `inline_recursion` is on, the function is expanded multiple times until it reaches the value set with the inline_depth pragma, the default value for recursive functions that is defined by the `inline_depth` pragma, or a capacity limit.

See also

Pragma directives and the __pragma and _Pragma keywords
inline_depth
/Ob (Inline function expansion)

`intrinsic` pragma

Article • 08/24/2022

Specifies that calls to functions specified in the pragma's argument list are intrinsic.

Syntax

> **`#pragma intrinsic(`** *function_1* [**`,`** *function_2* ...] **`)`**

Remarks

The **`intrinsic`** pragma tells the compiler that a function has known behavior. The compiler may call the function and not replace the function call with inline instructions, if it will result in better performance.

The library functions with intrinsic forms are listed below. Once an **`intrinsic`** pragma is seen, it takes effect at the first function definition containing a specified intrinsic function. The effect continues to the end of the source file or to the appearance of a **`function`** pragma specifying the same intrinsic function. The **`intrinsic`** pragma can be used only outside of a function definition, at the global level.

The following functions have intrinsic forms, and the intrinsic forms are used when you specify /Oi:

abs
_disable
_enable
fabs
_inp
_inpw

labs
_lrotl
_lrotr
memcmp
memcpy

memset
_outp
_outpw

_rotl
_rotr

strcat
strcmp
strcpy
strlen
_strset

Programs that use intrinsic functions are faster because they don't have the overhead of function calls. However, they may be larger because of the additional code generated.

x86-specific example

The `_disable` and `_enable` intrinsics generate kernel-mode instructions to disable or enable interrupts, and could be useful in kernel-mode drivers.

Compile the following code from the command line with `cl -c -FAs sample.c` and look at *`sample.asm`* to see that they turn into x86 instructions CLI and STI:

C++

```
// 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() {
}
```

Intrinsic floating-point functions

These floating-point functions don't have true intrinsic forms. Instead, they have versions that pass arguments directly to the floating-point chip, rather than pushing them on the stack:

acos
asin

cosh
fmod

pow
sinh

tanh

These floating-point functions have true intrinsic forms when you specify /Oi and /fp:fast (or any option that includes `/Oi`: /Ox, /O1, and /O2):

atan
atan2
cos

exp
log

log10
sin

sqrt
tan

You can use /fp:strict or /Za to override generation of true intrinsic floating-point options. In this case, the functions are generated as library routines that pass arguments directly to the floating-point chip instead of pushing them onto the program stack.

See #pragma function for information and an example on how to enable and disable intrinsics for a block of source text.

See also

Pragma directives and the __pragma and _Pragma keywords
Compiler intrinsics

`loop` pragma

Article • 08/03/2021

Controls how loop code is to be considered by the auto-parallelizer, or excludes a loop from consideration by the auto-vectorizer.

Syntax

> `#pragma loop( hint_parallel(` *n* `) )`
>
> `#pragma loop( no_vector )`
>
> `#pragma loop( ivdep )`

Parameters

`hint_parallel(` *n* `)`
A hint to the compiler that this loop should be parallelized across *n* threads, where *n* is a positive integer literal or zero. If *n* is zero, the maximum number of threads is used at run time. It's a hint to the compiler, not a command. There's no guarantee that the loop will be parallelized. If the loop has data dependencies, or structural issues, then it won't be parallelized. For example, it isn't parallelized if it stores to a scalar that's used beyond the loop body.

The compiler ignores this option unless the /Qpar compiler switch is specified.

`no_vector`
By default, the auto-vectorizer attempts to vectorize all loops that it evaluates may benefit from it. Specify this pragma to disable the auto-vectorizer for the loop that follows.

`ivdep`
A hint to the compiler to ignore vector dependencies for this loop.

Remarks

To use the `loop` pragma, place it immediately before, not in, a loop definition. The pragma takes effect for the scope of the loop that follows it. You can apply multiple pragma directives to a loop, in any order, but you must state each one in a separate pragma statement.

See also

Auto-parallelization and auto-vectorization

Pragma directives and the __pragma and _Pragma keywords

`make_public` pragma

Article • 08/03/2021

Indicates that a native type should have public assembly accessibility.

Syntax

> `#pragma make_public(` *type* `)`

Parameters

type
The name of the type you want to have public assembly accessibility.

Remarks

`make_public` is useful for when the native type you want to reference is from a header file that you can't change. If you want to use the native type in the signature of a public function in a type with public assembly visibility, the native type must also have public assembly accessibility, or the compiler will issue a warning.

`make_public` must be specified at global scope. It's only in effect from the point at which it's declared through to the end of the source code file.

The native type may be implicitly or explicitly private. For more information, see Type visibility.

Examples

The following sample is the contents of a header file that contains the definitions for two native structs.

C++

```
// make_public_pragma.h
struct Native_Struct_1 { int i; };
struct Native_Struct_2 { int i; };
```

The following code sample consumes the header file. It shows that, unless you explicitly mark the native structs as public by using `make_public`, the compiler will generate a warning when you attempt to use the native structs in the signature of public function in a public managed type.

C++

```
// 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
};
```

See also

Pragma directives and the __pragma and _Pragma keywords

`managed` and `unmanaged` pragma

Article • 08/03/2021

Enable function-level control to compile functions as managed or unmanaged.

Syntax

```
#pragma managed
#pragma unmanaged
#pragma managed( [ push, ] { on | off } )
#pragma managed(pop)
```

Remarks

The /clr compiler option provides module-level control for compiling functions either as managed or unmanaged.

An unmanaged function is compiled for the native platform. Execution of that portion of the program will be passed to the native platform by the common language runtime.

Functions are compiled as managed by default when `/clr` is used.

When applying a `managed` or `unmanaged` pragma:

- Add the pragma preceding a function, but not within a function body.
- Add the pragma after `#include` statements. Don't use it before any `#include` statements.

The compiler ignores the `managed` and `unmanaged` pragma if `/clr` isn't used in the compilation.

When a template function is instantiated, the pragma state when the template is defined determines if it's managed or unmanaged.

For more information, see Initialization of mixed assemblies.

Example

C++

```
// 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();
}
```

Output

```
In managed function.
In unmanaged function.
```

See also

Pragma directives and the __pragma and _Pragma keywords

`message` pragma

Article • 08/03/2021

Sends a string literal to the standard output without terminating the compilation.

Syntax

> **`#pragma message(`** *message-string* **`)`**

Remarks

A typical use of the **`message`** pragma is to display informational messages at compile time.

The *message-string* parameter can be a macro that expands to a string literal, and you can concatenate such macros with string literals in any combination.

If you use a predefined macro in the **`message`** pragma, the macro should return a string. Otherwise, you'll have to convert the output of the macro to a string.

The following code fragment uses the **`message`** pragma to display messages during compilation:

C++

```
// 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("")
```

See also

Pragma directives and the __pragma and _Pragma keywords

`omp` pragma

Article • 08/03/2021

Takes one or more OpenMP directives, along with any optional directive clauses.

Syntax

> `#pragma omp` *directive*

Remarks

For more information, see OpenMP directives.

See also

Pragma directives and the __pragma and _Pragma keywords

`once` pragma

Article • 08/03/2021

Specifies that the compiler includes the header file only once, when compiling a source code file.

Syntax

> `#pragma once`

Remarks

The use of `#pragma once` can reduce build times, as the compiler won't open and read the file again after the first `#include` of the file in the translation unit. It's called the *multiple-include optimization*. It has an effect similar to the *include guard* idiom, which uses preprocessor macro definitions to prevent multiple inclusions of the contents of the file. It also helps to prevent violations of the *one definition rule*: the requirement that all templates, types, functions, and objects have no more than one definition in your code.

For example:

C++

```
// header.h
#pragma once
// Code placed here is included only once per translation unit
```

We recommend the `#pragma once` directive for new code because it doesn't pollute the global namespace with a preprocessor symbol. It requires less typing, it's less distracting, and it can't cause *symbol collisions*. Symbol collisions are errors caused when different header files use the same preprocessor symbol as the guard value. It isn't part of the C++ Standard, but it's implemented portably by several common compilers.

There's no advantage to use of both the include guard idiom and `#pragma once` in the same file. The compiler recognizes the include guard idiom, and implements the multiple-include optimization the same way as the `#pragma once` directive if no non-comment code or preprocessor directive comes before or after the standard form of the idiom:

C++

```
// 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_
```

We recommend the include guard idiom when code must be portable to compilers that don't implement the `#pragma once` directive, to maintain consistency with existing code, or when the multiple-include optimization is impossible. It can occur in complex projects when file system aliasing or aliased include paths prevent the compiler from identifying identical include files by canonical path.

Be careful not to use `#pragma once` or the include guard idiom in header files designed to be included multiple times, that use preprocessor symbols to control their effects. For an example of this design, see the <assert.h> header file. Also be careful to manage your include paths to avoid creating multiple paths to included files, which can defeat the multiple-include optimization for both include guards and `#pragma once`.

See also

Pragma directives and the __pragma and _Pragma keywords

`optimize` pragma

Article • 03/01/2022

Specifies optimizations on a function-by-function basis.

Syntax

> **`#pragma optimize( "`** [*optimization-list*] **`",`** { **`on`** | **`off`** } **`)`**

Remarks

The **`optimize`** pragma must appear outside a function. It takes effect at the first function defined after the pragma is seen. The **`on`** and **`off`** arguments turn options specified in the *optimization-list* on or off.

The *optimization-list* can be zero or more of the parameters shown in the following table.

Parameters of the optimize Pragma

Parameter(s)	Type of optimization
`g`	Enable global optimizations. Deprecated. For more information, see /Og (Global optimizations).
`s` or **`t`**	Specify short or fast sequences of machine code.
`y`	Generate frame pointers on the program stack.

These parameters are the same letters used with the /O compiler options. For example, the following pragma is equivalent to the **`/Os`** compiler option:

```
C++

#pragma optimize( "s", on )
```

Using the **`optimize`** pragma with the empty string (**`""`**) is a special form of the directive:

When you use the **`off`** parameter, it turns all the optimizations, **`g`**, **`s`**, **`t`**, and **`y`**, off.

When you use the `on` parameter, it resets the optimizations to the ones that you specified using the /O compiler option.

```
C++
```

```
#pragma optimize( "", off )
/* unoptimized code section */
#pragma optimize( "", on )
```

See also

Pragma directives and the __pragma and _Pragma keywords

`pack` pragma

Article • 07/08/2022

Specifies the packing alignment for structure, union, and class members.

Syntax

```
#pragma pack( show )
#pragma pack( push [ , identifier ] [ , n ] )
#pragma pack( pop [ , { identifier | n } ] )
#pragma pack( [ n ] )
```

Parameters

`show`
(Optional) Displays the current byte value for packing alignment. The value is displayed by a warning message.

`push`
(Optional) Pushes the current packing alignment value on the internal compiler stack, and sets the current packing alignment value to *n*. If *n* isn't specified, the current packing alignment value is pushed.

`pop`
(Optional) Removes the record from the top of the internal compiler stack. If *n* isn't specified with **`pop`**, then the packing value associated with the resulting record on the top of the stack is the new packing alignment value. If *n* is specified, for example, `#pragma pack(pop, 16)`, *n* becomes the new packing alignment value. If you pop using an *`identifier`*, for example, `#pragma pack(pop, r1)`, then all records on the stack are popped until the record that has *`identifier`* is found. That record gets popped, and the packing value associated with the record found on the top of the stack becomes the new packing alignment value. If you pop using an *`identifier`* that isn't found in any record on the stack, then the **`pop`** is ignored.

The statement `#pragma pack (pop, r1, 2)` is equivalent to `#pragma pack (pop, r1)` followed by `#pragma pack(2)`.

`identifier`
(Optional) When used with **`push`**, assigns a name to the record on the internal compiler

stack. When used with `pop`, pops records off the internal stack until *`identifier`* is removed. If *`identifier`* isn't found on the internal stack, nothing is popped.

`n`
(Optional) Specifies the value, in bytes, to be used for packing. If the compiler option /Zp isn't set for the module, the default value for *`n`* is 8. Valid values are 1, 2, 4, 8, and 16. The alignment of a member is on a boundary that's either a multiple of *`n`*, or a multiple of the size of the member, whichever is smaller.

Remarks

To *pack* a class is to place its members directly after each other in memory. It can mean that some or all members can be aligned on a boundary smaller than the default alignment of the target architecture. `pack` gives control at the data-declaration level. It differs from compiler option /Zp, which only provides module-level control. **pack** takes effect at the first `struct`, `union`, or `class` declaration after the pragma is seen. `pack` has no effect on definitions. Calling `pack` with no arguments sets *`n`* to the value set in the compiler option `/Zp`. If the compiler option isn't set, the default value is 8 for x86, ARM, and ARM64. The default is 16 for x64 native and ARM64EC.

If you change the alignment of a structure, it may not use as much space in memory. However, you may see a loss of performance or even get a hardware-generated exception for unaligned access. You can modify this exception behavior by using SetErrorMode.

For more information about how to modify alignment, see these articles:

- alignof
- align
- __unaligned
- x64 structure alignment examples

> ⚠ **Warning**
>
> In Visual Studio 2015 and later you can use the standard `alignas` and `alignof` operators, which unlike `__alignof` and `__declspec( align )` are portable across compilers. The C++ standard doesn't address packing, so you must still use `pack` (or the corresponding extension on other compilers) to specify alignments smaller than the target architecture's word size.

Examples

The following sample shows how to use the `pack` pragma to change the alignment of a structure.

C++

```
// 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));
}
```

Output

```
0 4 8
0 4 6
```

The following sample shows how to use the *push*, *pop*, and *show* syntax.

C++

```
// 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
```

See also

Pragma directives and the __pragma and _Pragma keywords

`pointers_to_members` pragma

Article • 08/03/2021

C++ Specific

Specifies whether a pointer to a class member can be declared before its associated class definition. Used to control the pointer size, and the code required to interpret the pointer.

Syntax

```
#pragma pointers_to_members( best_case )
#pragma pointers_to_members( full_generality [ , most-general-representation ]
)
```

Remarks

You can place a `pointers_to_members` pragma in your source file as an alternative to using the /vmb or /vmg and /vmm, /vms, /vmv compiler options or the Microsoft-specific inheritance keywords.

The pointer-declaration argument specifies whether you've declared a pointer to a member before or after the associated function definition. The *`pointer-declaration`* argument is one of these two symbols:

- `full_generality`
  Generates safe, sometimes nonoptimal code. Use `full_generality` if any pointer to a member is declared before the associated class definition. This argument always uses the pointer representation specified by the *`most-general-representation`* argument. Equivalent to `/vmg`.

- `best_case`
  Generates optimal code using best-case representation for all pointers to members. Requires you to define the class before you declare a pointer to a member. The default is `best_case`.

The *`most-general-representation`* argument specifies the smallest pointer representation that the compiler should use to reference safely any pointer to a member of a class in a translation unit. The argument can be one of these values:

- `single_inheritance`
  The most general representation is single-inheritance pointer to member function. Equivalent to `/vmg /vms`. Causes an error if the inheritance model of a class definition is either multiple or virtual.

- `multiple_inheritance`
  The most general representation is multiple-inheritance pointer to member function. Equivalent to `/vmg /vmm`. Causes an error if the inheritance model of a class definition is virtual.

- `virtual_inheritance`
  The most general representation is virtual-inheritance pointer to member function. Equivalent to `/vmg /vmv`. Never causes an error. `virtual_inheritance` is the default argument when `#pragma pointers_to_members(full_generality)` is used.

> **⊗ Caution**
>
> We advise you to put the `pointers_to_members` pragma only in the source code file that you want to affect, and only after any `#include` directives. This practice reduces the risk that the pragma will affect other files, and that you'll accidentally specify multiple definitions for the same variable, function, or class name.

Example

C++

```
//   Specify single-inheritance only
#pragma pointers_to_members( full_generality, single_inheritance )
```

END C++ Specific

See also

Pragma directives and the __pragma and _Pragma keywords

`pop_macro` pragma

Article • 08/03/2021

Sets the value of the *macro-name* macro to the value on the top of the stack for this macro.

Syntax

> `#pragma pop_macro(` "*macro-name*" `)`

Remarks

A push_macro for *macro-name* must be issued before you can do a `pop_macro`.

Example

C++

```
// 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);
}
```

Output

```
1
2
1
```

```
2
1
3
```

See also

Pragma directives and the __pragma and _Pragma keywords

`push_macro` pragma

Article • 08/03/2021

Saves the value of the *macro-name* macro on the top of the stack for this macro.

Syntax

> `#pragma push_macro("` *macro-name* `")`

Remarks

You can retrieve the value for *macro-name* with `pop_macro`.

See pop_macro pragma for a sample.

See also

Pragma directives and the __pragma and _Pragma keywords

`region` and `endregion` pragma

Article • 08/03/2021

`#pragma region` lets you specify a block of code that you can expand or collapse when using the outlining feature of the Visual Studio editor.

Syntax

> **`#pragma region`** *name*
>
> **`#pragma endregion`** *comment*

Parameters

comment
(Optional) A comment to display in the code editor.

name
(Optional) The name of the region. This name displays in the code editor.

Remarks

`#pragma endregion` marks the end of a `#pragma region` block.

A `#pragma region` block must be terminated by a `#pragma endregion` directive.

Example

C++

```
// pragma_directives_region.cpp
#pragma region Region_1
void Test() {}
void Test2() {}
void Test3() {}
#pragma endregion Region_1

int main() {}
```

See also

Pragma directives and the __pragma and _Pragma keywords

`runtime_checks` pragma

Article • 08/03/2021

Disables or restores the /RTC compiler option settings.

Syntax

> **`#pragma runtime_checks( "`** [*runtime-check-options*] **`",`** { **`restore`** | **`off`** } **`)`**

Remarks

You can't enable a run-time check that wasn't enabled by a compiler option. For example, if you don't specify **`/RTCs`** on the command line, specifying `#pragma runtime_checks( "s", restore)` won't enable stack frame verification.

The **`runtime_checks`** pragma must appear outside a function, and takes effect at the first function defined after the pragma is seen. The **`restore`** and **`off`** arguments turn options specified in the **`runtime_checks`** pragma on or off.

The *runtime-check-options* can be zero or more of the parameters shown in the following table.

Parameters of the runtime_checks Pragma

Parameter(s)	Type of run-time check
`s`	Enables stack (frame) verification.
`c`	Reports when a value is assigned to a smaller data type that results in a data loss.
`u`	Reports when a variable is used before it's defined.

These parameters are the same ones used with the **`/RTC`** compiler option. For example:

```
C++

#pragma runtime_checks( "sc", restore )
```

Using the **`runtime_checks`** pragma with the empty string (**`""`**) is a special form of the directive:

- When you use the `off` parameter, it turns the run-time error checks listed in the table above, off.
- When you use the `restore` parameter, it resets the run-time error checks to the ones that you specified using the `/RTC` compiler option.

C++

```
#pragma runtime_checks( "", off )
/* runtime checks are off in this region */
#pragma runtime_checks( "", restore )
```

See also

Pragma directives and the __pragma and _Pragma keywords

`section` pragma

Article • 08/03/2021

Creates a section in an OBJ file.

Syntax

> `#pragma section( "` *section-name* `"` [`,` *attributes*] `)`

Remarks

The terms *segment* and *section* have the same meaning in this article.

Once a section is defined, it remains valid for the rest of the compilation. However, you must use __declspec(allocate), or nothing is placed in the section.

section-name is a required parameter that becomes the name of the section. The name must not conflict with any standard section names. See /SECTION for a list of names you shouldn't use when creating a section.

attributes is an optional parameter consisting of one or more comma-separated attributes to assign to the section. Possible *attributes* are:

Attribute	Description
`read`	Allows read operations on data.
`write`	Allows write operations on data.
`execute`	Allows code to be executed.
`shared`	Shares the section among all processes that load the image.
`nopage`	Marks the section as not pageable. Useful for Win32 device drivers.
`nocache`	Marks the section as not cacheable. Useful for Win32 device drivers.
`discard`	Marks the section as discardable. Useful for Win32 device drivers.
`remove`	Marks the section as not memory-resident. For virtual device drivers (V*x*D) only.

If you don't specify any attributes, the section has `read` and `write` attributes.

Example

In this example, the first section pragma identifies the section and its attributes. The integer `j` isn't put into `mysec` because it wasn't declared using `__declspec(allocate)`. Instead, `j` goes into the data section. The integer `i` does go into `mysec` because of its `__declspec(allocate)` storage-class attribute.

C++

```
// pragma_section.cpp
#pragma section("mysec",read,write)
int j = 0;

__declspec(allocate("mysec"))
int i = 0;

int main(){}
```

See also

Pragma directives and the __pragma and _Pragma keywords

`setlocale` pragma

Article • 11/08/2022

Defines the *locale*, the country/region and language to use when translating wide-character constants and string literals.

Syntax

> **`#pragma setlocale( "`** [*locale-string*] **`" )`**

Remarks

Because the algorithm for converting multibyte characters to wide characters may vary by locale, or the compilation may take place in a different locale from where an executable file will be run, this pragma provides a way to specify the target locale at compile time. It guarantees wide-character strings are stored in the correct format.

The default *locale-string* is the empty string, specified by `#pragma setlocale( "" )`.

The **`"C"`** locale maps each character in the string to its value as a **`wchar_t`**. Other valid values for `setlocale` are the entries found in the Language strings list. For example, you could specify:

```
C++
```

```
#pragma setlocale("dutch")
```

The ability to specify a language string depends on the code page and language ID support on your computer.

See also

Pragma directives and the __pragma and _Pragma keywords

`strict_gs_check` pragma

Article • 08/03/2021

This pragma provides enhanced security checking.

Syntax

> `#pragma strict_gs_check(` [`push,`] { `on` | `off` } `)`
>
> `#pragma strict_gs_check( pop )`

Remarks

Instructs the compiler to insert a random cookie in the function stack to help detect some categories of stack-based buffer overrun. By default, the `/GS` compiler option doesn't insert a cookie for all functions. For more information, see /GS (Buffer Security Check).

Compile by using `/GS` to enable `strict_gs_check`.

Use this pragma in code modules that are exposed to potentially harmful data. `strict_gs_check` is an aggressive pragma, and is applied to functions that might not need this defense, but is optimized to minimize its effect on the performance of the resulting application.

Even if you use this pragma, you should strive to write secure code. That is, make sure that your code has no buffer overruns. `strict_gs_check` might protect your application from buffer overruns that do remain in your code.

Example

In this sample, a buffer overrun occurs when we copy an array to a local array. When you compile this code with `/GS`, no cookie is inserted in the stack, because the array data type is a pointer. Adding the `strict_gs_check` pragma forces the stack cookie into the function stack.

C++

```
// 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;
}
```

See also

Pragma directives and the __pragma and _Pragma keywords

/GS (Buffer security check)

`system_header` pragma

Article • 08/03/2021

Treat the rest of the file as external for diagnostics reports.

Syntax

```
#pragma system_header
```

Remarks

The `system_header` pragma tells the compiler to show diagnostics at the level specified by the `/external:Wn` option for the rest of the current source file. For more information on how to specify external files and the external warning level to the compiler, see /external.

The `system_header` pragma doesn't apply past the end of the current source file. In other words, it doesn't apply to files that include this file. The `system_header` pragma applies even if no other files are specified as external to the compiler. However, if no `/external:Wn` option level is specified, the compiler may issue a diagnostic and uses the same warning level it applies to non-external files. Other pragma directives that affect warning behavior still apply after a `system_header` pragma. The effect of `#pragma system_header` is similar to the warning pragma:

C++

```
// 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 )
```

The `system_header` pragma is available starting in Visual Studio 2019 version 16.10.

Example

This sample header shows how to mark the contents of a file as external:

C++

```
// 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
```

See also

/external
warning pragma
/Wn (Compiler warning level)
Pragma directives and the __pragma and _Pragma keywords

`vtordisp` pragma

Article • 08/03/2021

Controls the addition of the hidden `vtordisp` construction/destruction displacement member. The `vtordisp` pragma is C++-specific.

Syntax

> **`#pragma vtordisp(`** [**`push,`**] *n* **`)`**
>
> **`#pragma vtordisp(pop)`**
>
> **`#pragma vtordisp()`**
>
> **`#pragma vtordisp(`** [**`push,`**] { **`on`** | **`off`** } **`)`**

Parameters

`push`

Pushes the current `vtordisp` setting on the internal compiler stack, and sets the new `vtordisp` setting to *n*. If *n* isn't specified, the current `vtordisp` setting is unchanged.

`pop`

Removes the top record from the internal compiler stack, and restores the `vtordisp` setting to the removed value.

n

Specifies the new value for the `vtordisp` setting. Possible values are **`0`**, **`1`**, or **`2`**, corresponding to the **`/vd0`**, **`/vd1`**, and **`/vd2`** compiler options. For more information, see /vd (Disable Construction Displacements).

`on`

Equivalent to `#pragma vtordisp(1)`.

`off`

Equivalent to `#pragma vtordisp(0)`.

Remarks

The **`vtordisp`** pragma is applicable only to code that uses virtual bases. If a derived class overrides a virtual function that it inherits from a virtual base class, and if a constructor or destructor for the derived class calls that function using a pointer to the virtual base

class, the compiler might introduce extra hidden `vtordisp` fields into classes with virtual bases.

The **`vtordisp`** pragma affects the layout of classes that follow it. The **`/vd0`**, **`/vd1`**, and **`/vd2`** compiler options specify the same behavior for complete modules. Specifying **`0`** or **`off`** suppresses the hidden `vtordisp` members. Turn off **`vtordisp`** only if there's no possibility that the class's constructors and destructors call virtual functions on the object pointed to by the **`this`** pointer.

Specifying **`1`** or **`on`**, the default, enables the hidden `vtordisp` members where they're necessary.

Specifying **`2`** enables the hidden `vtordisp` members for all virtual bases with virtual functions. `#pragma vtordisp(2)` might be necessary to ensure correct performance of **`dynamic_cast`** on a partially constructed object. For more information, see Compiler Warning (level 1) C4436.

`#pragma vtordisp()`, with no arguments, restores the `vtordisp` setting to its initial setting.

C++

```
#pragma vtordisp(push, 2)
class GetReal : virtual public VBase { ... };
#pragma vtordisp(pop)
```

See also

Pragma directives and the __pragma and _Pragma keywords

`warning` pragma

Article • 01/25/2023

Enables selective modification of the behavior of compiler warning messages.

Syntax

```
#pragma warning(
    warning-specifier : warning-number-list
    [; warning-specifier : warning-number-list ... ] )
#pragma warning( push [ , n ] )
#pragma warning( pop )
```

Remarks

The following warning-specifier parameters are available.

warning-specifier	Meaning
`1`, `2`, `3`, `4`	Apply the given level to the specified warnings. Also turns on a specified warning that is off by default.
`default`	Reset warning behavior to its default value. Also turns on a specified warning that is off by default. The warning will be generated at its default, documented, level. For more information, see Compiler warnings that are off by default.
`disable`	Don't issue the specified warning messages.
`error`	Report the specified warnings as errors.
`once`	Display the specified message(s) only one time.
`suppress`	Pushes the current state of the pragma on the stack, disables the specified warning for the next line, and then pops the warning stack so that the pragma state is reset.

The following code statement illustrates that a *`warning-number-list`* parameter can contain multiple warning numbers, and that multiple *`warning-specifier`* parameters can be specified in the same pragma directive.

```
C++
```

```
#pragma warning( disable : 4507 34; once : 4385; error : 164 )
```

This directive is functionally equivalent to the following code:

C++

```
// 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 )
```

The compiler adds 4000 to any warning number that is between 0 and 999.

Warning numbers in the range 4700-4999 are associated with code generation. For these warnings, the state of the warning in effect when the compiler reaches the function definition remains in effect for the rest of the function. Use of the **`warning`** pragma in the function to change the state of a warning number larger than 4699 only takes effect after the end of the function. The following example shows the correct placement of a **`warning`** pragma to disable a code-generation warning message, and then to restore it.

C++

```
// 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
}
```

Notice that throughout a function body, the last setting of the **`warning`** pragma will be in effect for the whole function.

Push and pop

The `warning` pragma also supports the following syntax, where the optional *n* parameter represents a warning level (1 through 4).

```
#pragma warning( push [ , n ] )
```

```
#pragma warning( pop )
```

The pragma `warning( push )` stores the current warning state for every warning. The pragma `warning( push, n )` stores the current state for every warning and sets the global warning level to *n*.

The pragma `warning( pop )` pops the last warning state pushed onto the stack. Any changes that you made to the warning state between `push` and `pop` are undone. Consider this example:

C++

```
#pragma warning( push )
#pragma warning( disable : 4705 )
#pragma warning( disable : 4706 )
#pragma warning( disable : 4707 )
// Some code
#pragma warning( pop )
```

At the end of this code, `pop` restores the state of every warning (includes 4705, 4706, and 4707) to what it was at the start of the code.

When you write header files, you can use `push` and `pop` to guarantee that warning-state changes made by a user don't prevent the headers from compiling correctly. Use `push` at the start of the header and `pop` at the end. For example, you may have a header that doesn't compile cleanly at warning level 4. The following code changes the warning level to 3, and then restores the original warning level at the end of the header.

C++

```
#pragma warning( push, 3 )
// Declarations/definitions
#pragma warning( pop )
```

For more information about compiler options that help you suppress warnings, see /FI and /w.

See also

Pragma directives and the __pragma and _Pragma keywords