Keywords for Generics

Generic Functions

[attributes] [modifiers]

return-type identifier <type-parameter identifier(s)> [type-parameter-constraints clauses]

([formal-parameters])

{

function-body

}

Parameters

attributes (Optional)

Additional declarative information. For more information on attributes and attribute classes, see attributes.

modifiers (Optional)

A modifier for the function, such as static. virtual is not allowed since virtual methods may not be generic.

return-type

The type returned by the method. If the return type is void, no return value is required.

identifier

The function name.

type-parameter identifier(s)

Comma-separated identifiers list.

formal-parameters (Optional)

Parameter list.

type-parameter-constraints-clauses

This specifies restrictions on the types that may be used as type arguments, and takes the form specified in Constraints.

function-body

The body of the method, which may refer to the type parameter identifiers.

Code	MD-UML
`generic <typename ItemType>` `void G(int i) {}`	ItemType with «C++TemplateParameter» applied and set tagged value **type keyword** = typename.

`ref struct MyStruct {`
`generic <typename Type1>`
`void G(Type1 i) {}`
`generic <typename Type2>`
`static void H(int i) {}`
`};`	

Generic Classes

[attributes]

generic <class-key type-parameter-identifier(s)**>**

[constraint-clauses]

[accessibility-modifiers] **ref class** identifier [modifiers] [: base-list]

{

class-body

} [declarators] [;]

Parameters

attributes (optional)

Additional declarative information. For more information on attributes and attribute classes, see Attributes.

class-key

Either class or typename

type-parameter-identifier(s)

Comma-separated list of identifiers specifying the names of the type parameters.

constraint-clauses

A list (not comma-separated) of where clauses specifying the constraints for the type parameters. Takes the form:

where type-parameter-identifier : constraint-list ...

constraint-list

class-or-interface[, ...]

accessibility-modifiers

Allowed accessibility modifiers include public and private. identifier

The name of the generic class, any valid C++ identifier.

modifiers (optional)

Allowed modifiers include sealed and abstract.

base-list

A list that contains the one base class and any implemented interfaces, all separated by commas.

class-body

The body of the class, containing fields, member functions, etc.

declarators

Declarations of any variables of this type. For example: ^identifier[, ...]

Code	MD-UML

`interface class MyInterface {};`
`ref class MyBase{};`
`generic <class T1, class T2>`
`where T1 : MyInterface, MyBase`
`where T2 : MyBase`
`ref class MyClass {};`	

Generic Interfaces

[attributes] **generic <**class-key type-parameter-identifier[, ...]**>**

[type-parameter-constraints-clauses]

[accesibility-modifiers] **interface class** identifier [: base-list] { interface-body} [declarators] ;

Parameters

attributes (optional)

Additional declarative information. For more information on attributes and attribute classes, see Attributes.

class-key

class or typename

type-parameter-identifier(s)

Comma-separated identifiers list.

type-parameter-constraints-clauses

Takes the form specified in Constraints

accessibility-modifiers (optional)

Accessibility modifiers (e.g. public, private).

identifier

The interface name.

base-list (optional)

A list that contains one or more explicit base interfaces separated by commas.

interface-body

Declarations of the interface members.

declarators (optional)

Declarations of variables based on this type.

Code	MD-UML

`generic <typename Itype1,`
`typename Itype2>`
`public interface class List {`
`  Itype1 x;`
`};`
`generic <typename ItemType>`
`ref class List2 : public`
`List<ItemType, char>`
`{`
`};`	

Generic Delegates

[attributes]

generic < [class | typename] type-parameter-identifiers >

[type-parameter-constraints-clauses]

[accessibility-modifiers] delegate result-type identifier

([formal-parameters]);

Parameters

attributes (Optional)

Additional declarative information. For more information on attributes and attribute classes, see Attributes.

type-parameter-identifier(s)

Comma-separated list of identifiers for the type parameters.

type-parameter-constraints-clauses

Takes the form specified in Constraints

accessibility-modifiers (Optional)

Accessibility modifiers (e.g., public, private).

result-type

The return type of the delegate.

identifier

The name of the delegate.

formal-parameters (Optional)

The parameter list of the delegate.

Code	MD-UML
`generic < class IT>` `delegate IT GenDelegate(IT p1, IT% p2);`	

Clr Data Member option

initonly

---

initonly indicates that variable assignment can only occur as part of the declaration or in a static constructor in the same class.

Code	MD-UML

`ref struct MyStruct {`
`  initonly`
`  static int staticConst1;`
`  initonly`
`  static int staticConst2 = 2;`
`  static MyStruct() {`
`  staticConst1 = 1;`
`  }`
`};`
	MyStruct with «C++CLIStruct»
	staticConst1 and staticConst2 have «C++CLIAttribute» applied with tagged value field = initonly and Is Static = true

literal

A variable (data member) marked as literal in a /clr compilation is the native equivalent of a static const variable.

A data member marked as literal must be initialized when declared and the value must be a constant integral, enum, or string type. Conversion from the type of the initialization expression to the type of the static const data-member must not require a user-defined conversion.

Code	MD-UML

`ref class MyClassWithLiteral`
 	
`{`
 	
`literal int i = 1;`
 	
`};`
	MyClassWithLiteral with «C++CLIClass» Attribute i has «C++CLIAttribute» applied with tagged value field = literal and set to Is Static = true and Is Read Only = true

Inheritance Keywords

__single_inheritance, __multiple_inheritance, __virtual_inheritance

Grammar:

class [__single_inheritance] class-name;

class [__multiple_inheritance] class-name;

class [__virtual_inheritance] class-name;

Parameter

class-name

The name of the class being declared.

Code	MD-UML
`class __single_inheritance S;`	Class with «VC++Class»

Microsoft-Specific Native declaration keywords

__interface

Grammar:

modifier __interface interface-name {interface-definition};

A Visual C++ interface can be defined as follows:

- Can inherit from zero or more base interfaces.
- Cannot inherit from a base class.
- Can only contain public, pure virtual methods.
- Cannot contain constructors, destructors, or operators.
- Cannot contain static methods.
- Cannot contain data members; properties are allowed.

Code	MD-UML
`__interface MyInterface {};`	

__delegate

Grammar:

__delegate function-declarator

A delegate is roughly equivalent to a C++ function pointer except for the following difference:

- A delegate can only be bound to one or more methods within a __gc class.

When the compiler encounters the __delegate keyword, a definition of a __gc class is generated. This __gc class has the following characteristics:

- It inherits from System::MulticastDelegate.
- It has a constructor that takes two arguments: a pointer to a __gc class or NULL (in the case of binding to a static method) and a fully qualified method of the specified type.
- It has a method called Invoke, whose signature matches the declared signature of the delegate.

Code	MD-UML

`__delegate int MyDelegate();`	

__event

Grammar:

- __event method-declarator;
- __event __interface interface-specifier;
- __event member-declarator;

Native Events

Code	MD-UML

`class Source {`
`public:`
`  __event void MyEvent(int i);`
`};`	

Com Events

The __interface keyword is always required after __event for a COM event source.

Code	MD-UML

`__interface IEvents {`
`  };`
`class CSource {`
`public:`
`  __event __interface IEvents;`
`};`
	__event __interface IEvents;
	is mapped to attribute without name

NOTE This mapping produces the exceptional case for syntax checker. The syntax checker has to allow attribute without name for this case.

Managed Events

Code	MD-UML

`__interface IEvents {`
`  };`
`class CSource {`
`public:`
`  __event __interface IEvents;`
`};`
	__event __interface IEvents;
	is mapped to attribute without name

public __delegate void D();

public __gc class X { public:

__event D* E;

__event void noE(); };

NOTE __delegate is in C++ Managed Profile whereas __event is in Microsoft Visual C++ Profile.

Microsoft-Specific Modifiers

Based addressing with __based

The __**based** keyword allows you to declare pointers based on pointers (pointers that are offsets from existing pointers).

type __**based**(base) declarator

Example

// based_pointers1.cpp

// compile with: /c

void *vpBuffer;

struct llist_t {

    void __based(vpBuffer) *vpData;

    struct llist_t __based(vpBuffer) *llNext;

};

Code	MD-UML
`void *vpBuffer;` `void __based(vpBuffer) *vpData;`	

Function calling conventions

The Visual C/C++ compiler provides several different conventions for calling internal and external functions.

__**cdecl**

return-type __**cdecl** function-name[(argument-list)]

This is the default calling convention for C and C++ programs. Because the stack is cleaned up by the caller, it can do **vararg** functions. The __**cdecl** calling convention creates larger executables than __**stdcall**, because it requires each function call to include stack cleanup code. The following list shows the implementation of this calling convention.

Code	MD-UML

`class CMyClass {`
`void __cdecl myMethod();`
`};`
	__event __interface IEvents;
	is mapped to attribute without name

__**clrcall**

return-type __**clrcall** function-name[(argument-list)]

Specifies that a function can only be called from managed code. Use __**clrcall** for all virtual functions that will only be called from managed code. However, this calling convention cannot be used for functions that will be called from native code.

Example

// compile with: /clr:oldSyntax /LD

void __clrcall Test1( ) {}

void (__clrcall *fpTest1)( ) = &Test1;

Code	MD-UML

`__interface IEvents {`
`  };`
`class CSource {`
`public:`
`   __event __interface IEvents;`
`};`
	__event __interface IEvents;
	is mapped to attribute without name

Code

void __clrcall Test1( ) {} void (__clrcall *fpTest1)( ) = &Test1;

MD-UML

217

__stdcall

return-type __stdcall function-name[(argument-list)]

The __stdcall calling convention is used to call Win32 API functions. The callee cleans the stack, so the com- piler makes vararg functions __cdecl. Functions that use this calling convention require a function prototype.

Code	MD-UML

`__interface IEvents {`
`  };`
`class CSource {`
`public:`
`   __event __interface IEvents;`
`};`
	__event __interface IEvents;
	is mapped to attribute without name

Code MD-UML

class CmyClass2 {
void __stdcall mymethod();

};

__fastcall
return-type __fastcall function-name[(argument-list)]

The __fastcall calling convention specifies that arguments to functions are to be passed in registers, when possible. The following list shows the implementation of this calling convention.

Example

class CmyClass3 {
void __fastcall mymethod();

};

Code	MD-UML

`__interface IEvents {`
`  };`
`class CSource {`
`public:`
`   __event __interface IEvents;`
`};`
	__event __interface IEvents;
	is mapped to attribute without name

Code MD-UML

class CmyClass3 {
void __fastcall mymethod();

};

__thiscall
return-type __thiscall function-name[(argument-list)]

The __thiscall calling convention is used on member functions and is the default calling convention used by C++ member functions that do not use variable arguments. Under __thiscall, the callee cleans the stack, which is impossible for vararg functions. Arguments are pushed on the stack from right to left, with the this pointer being passed via register ECX, and not on the stack, on the x86 architecture.

Example

// compile with: /c /clr:oldSyntax class CmyClass4 {

void __thiscall mymethod();

void __clrcall mymethod2(); };

Code	MD-UML

`__interface IEvents {`
`  };`
`class CSource {`
`public:`
`   __event __interface IEvents;`
`};`
	__event __interface IEvents;
	is mapped to attribute without name

Code

class CmyClass4 {
void __thiscall mymethod();

};

MD-UML

__unaligned
type __unaligned pointer_identifier

When a pointer is declared as __unaligned, the compiler assumes that the type or data pointed to is not aligned. __unaligned is only valid in compilers for x64 and the Itanium Processor Family (IPF).

Example

// compile with: /c // processor: x64 IPF #include <stdio.h> int main() {

char buf[100];
int __unaligned *p1 = (int*)(&buf[37]); int *p2 = (int *)p1;
*p1=0; //ok
__try {

*p2 = 0; // throws an exception }

Code	MD-UML

`__interface IEvents {`
`  };`
`class CSource {`
`public:`
`   __event __interface IEvents;`
`};`
	__event __interface IEvents;
	is mapped to attribute without name

Code

__except(1) { puts("exception");

}}

int __unaligned *p1;

MD-UML

__w64

type __w64 identifier

Parameters

type

One of the three types that might cause problems in code being ported from a 32-bit to a 64-bit compiler: int, long, or a pointer.

identifier
The identifier for the variable you are creating.

__w64 lets you mark variables, such that when you compile with /Wp64 the compiler will report any warnings that would be reported if you were compiling with a 64-bit compiler.

Example

// compile with: /W3 /Wp64 typedef int Int_32;
#ifdef _WIN64
typedef __int64 Int_Native; #else

typedef int __w64 Int_Native; #endif
int main() {

Int_32 i0 = 5;
Int_Native i1 = 10;
i0 = i1; // C4244 64-bit int assigned to 32-bit int

// char __w64 c; error, cannot use __w64 on char }

Code	MD-UML

`__interface IEvents {`
`  };`
`class CSource {`
`public:`
`   __event __interface IEvents;`
`};`
	__event __interface IEvents;
	is mapped to attribute without name

Code

int __w64 i;

Extended storage-class attributes with __declspec

The extended attribute syntax for specifying storage-class information uses the __declspec keyword, which specifies that an instance of a given type is to be stored with a Microsoft-specific storage-class attribute listed below. Examples of other storage-class modifiers include the static and extern keywords. However, these key- words are part of the ANSI specification of the C and C++ languages, and as such are not covered by extended attribute syntax. The extended attribute syntax simplifies and standardizes Microsoft-specific extensions to the C and C++ languages.

decl-specifier:
__declspec (extended-decl-modifier-seq) extended-decl-modifier-seq: extended-decl-modifieropt
extended-decl-modifier extended-decl-modifier-seq extended-decl-modifier:
align(#)
allocate("segname")
appdomain
deprecated
dllimport
dllexport
jitintrinsic
naked

MD-UML

noalias
noinline
noreturn
nothrow
novtable
process property({get=get_func_name|,put=put_func_name}) restrict

selectany
thread uuid("ComObjectGUID")

White space separates the declaration modifier sequence.

Extended attribute grammar supports these Microsoft-specific storage-class attributes: align, allocate, appdo- main, deprecated, dllexport, dllimport, jitintrin sic, naked, noalias, noinline, noreturn, nothrow, novtable, process, restrict, selectany, and thread. It also supports these COM-object attributes: property an d uuid.

The dllexport, dllimport, naked, noalias, nothrow, property, restrict, selectany, thread, and uuid storage- class attributes are properties only of the declaration of the object or function to which they are applied. The thread attribute affects data and objects only. The naked attribute affects functions only. The dllimport and dll- export attributes affect functions, data, and objects. The property, selectany, and uuid attributes affect COM objects.

The __declspec keywords should be placed at the beginning of a simple declaration. The compiler ignores, without warning, any __declspec keywords placed after * or & and in front of the variable identifier in a declara- tion.

Note:

- A __declspec attribute specified in the beginning of a user-defined type declaration applies to the variable of that type.
- A __declspec attribute placed after the class or struct keyword applies to the user-defined type.

Code	MD-UML

`__interface IEvents {`
`  };`
`class CSource {`
`public:`
`   __event __interface IEvents;`
`};`
	__event __interface IEvents;
	is mapped to attribute without name

Code MD-UML

class __declspec(dllimport) X {};

Code	MD-UML

`__interface IEvents {`
`  };`
`class CSource {`
`public:`
`   __event __interface IEvents;`
`};`
	__event __interface IEvents;
	is mapped to attribute without name

Code MD-UML

__declspec(dllimport) class X {} varX;

__restrict

The __restrict keyword is valid only on variables, and __declspec(restrict) is only valid on function declara- tions and definitions.

When __restrict is used, the compiler will not propagate the no-alias property of a variable. That is, if you assign a __restrict variable to a non-__restrict vari able, the compiler will not imply that the non-__restrict variable is not aliased.

Generally, if you affect the behavior of an entire function, it is better to use the __declspec than the keyword. __restrict is similar to restrict from the C99 spec, but __restrict can be used in C++ or C programs.

Example

// __restrict_keyword.c
// compile with: /LD
// In the following function, declare a and b as disjoint arrays // but do not have same assurance for c and d.
void sum2(int n, int * __restrict a, int * __restrict b,

int * c, int * d) { int i;

for (i = 0; i < n; i++) { a[i] = b[i] + c[i]; c[i] = b[i] + d[i];

} }

- // By marking union members as __restrict, tells the compiler that
- // only z.x or z.y will be accessed in any given scope. union z {

int * __restrict x;

double * __restrict y; };

Code	MD-UML

`__interface IEvents {`
`  };`
`class CSource {`
`public:`
`   __event __interface IEvents;`
`};`
	__event __interface IEvents;
	is mapped to attribute without name

Code

MD-UML

int * __restrict x;

__forceinline, __inline

The insertion (called inline expansion or inlining) occurs only if the compiler's cost/benefit analysis show it to be profitable. Inline expansion alleviates the function-call overhead at the potential cost of larger code size.

The __forceinline keyword overrides the cost/benefit analysis and relies on the judgment of the programmer instead. Exercise caution when using __forcein line. Indiscriminate use of __forceinline can result in larger code with only marginal performance gains or, in some cases, even performance losses (due to increased pag- ing of a larger executable, for example).

Using inline functions can make your program faster because they eliminate the overhead associated with function calls. Functions expanded inline are subject to code optimizations not available to normal functions.

The compiler treats the inline expansion options and keywords as suggestions. There is no guarantee that functions will be inlined. You cannot force the compiler to inline a particular function, even with the __forcein- line keyword. When compiling with /clr, the compiler will not inline a function if there are security attributes applied to the function.

The inline keyword is available only in C++. The __inline and __forceinline keywords are available in both C and C++. For compatibility with previous versions, _inline is a synonym for __inline.

Grammar:

__inline function_declarator; __forceinline function_declarator;

Code	MD-UML

`__interface IEvents {`
`  };`
`class CSource {`
`public:`
`   __event __interface IEvents;`
`};`
	__event __interface IEvents;
	is mapped to attribute without name

Code

__inline int max(int a , int b) { if(a > b)

`return a; return b;`

}

MD-UML

C++ Attributes

Attributes are designed to simplify COM programming and .NET Framework common language runtime development. When you include attributes in your source files, the compiler works with provider DLLs to insert code or modify the code in the generated object files.

Code	MD-UML

`[ coclass,`
`aggregatable(allowed),`
`  uuid("1a8369cc-1c91-`
`42c4-befa-5a5d8c9d2529")]`
`class CMyClass {};`
	__event __interface IEvents;
	is mapped to attribute without name