

## 微软VC++的\_\_declspec关键字用法

\_\_declspec用于指定所给定类型的实例的与Microsoft相关的存储方式。其它的有关存储方式的修饰符如static与extern等是C和C++语言的ANSI规范，而\_\_declspec是一种扩展属性的定义。扩展属性语法简化并标准化了C和C++语言关于Microsoft的扩展。

用法：\_\_declspec ( extended-decl-modifier )

extended-decl-modifier参数如下，可同时出现，中间有空格隔开：

align ( C++ )  
allocate  
appdomain  
deprecated ( C++ )  
dllimport  
dllexport  
jitintrinsic  
naked ( C++ )  
noalias  
noinline  
noreturn  
nothrow ( C++ )  
novtable  
process  
property ( C++ )  
restrict  
selectany  
thread  
uuid ( C++ )

1.\_\_declspec关键字应该出现在简单声明的前面。对于出现在\*或&后面或者变量声明中标识符的前面的\_\_declspec，编译器将忽略并且不给出警告。

2.要注意区分\_\_declspec是修饰类型还是修饰变量：

\_\_declspec(align(8)) struct Str b;修饰的是变量b。其它地方定义的struct Str类型的变量将不受\_\_declspec(align(8))影响。

\_\_declspec(align(8)) struct Str {};修饰的是struct Str类型。所有该类型的变量都受\_\_declspec(align(8))影响。

align：

格式：\_\_declspec(align(n)) declarator

其中，n是对齐参数，其有效值是2的整数次幂（从1到8192字节），如2，4，8，16，32或64。参数declarator是要设置对齐方式的数据。

1.使用\_\_declspec(align(n))来精确控制用户自定义数据的对齐方式。你可以在定义struct，union，class或声明变量时使用\_\_declspec(align(n))。

2.不能为函数参数使用\_\_declspec(align(n))。

- 3.如果未使用\_\_declspec(align(#)), 编译器将根据数据大小按自然边界对齐。如4字节整数按4字节边界对齐; 8字节double按8字节边界对齐。类或结构体中的数据, 将取数据本身自然对齐方式和#pragma pack(n)设置的对齐系数中的最小值进行对齐。
- 4.\_\_declspec(align(n))和#pragma pack(n)是一对兄弟, 前者规定了对齐系数的最小值, 后者规定了对齐系数的最大值。
- 5.当两者同时出现时, 前者拥有更高的优先级。即, 当两者同时出现且值矛盾时, 后者将不起作用。
- 6.当变量size大于等于#pragma pack(n)指定的n, 而且\_\_declspec(align(n))指定的数值n比对应类型长度小的时候, 这个\_\_declspec(align(n))指定将不起作用。
- 7.当#pragma pack(n)指定的值n大于等于所有数据成员size的时候, 这个值n将不起作用。

allocate :

格式: \_\_declspec(allocate("segname")) declarator

为数据指定存储的数据段。数据段名必须为以下列举中的一个:

code\_seg

const\_seg

data\_seg

init\_seg

section

appdomain :

指定托管程序中的每个应用程序域都要有一份指定全局变量或静态成员变量的拷贝。

deprecated :

与#pragma deprecated()的作用相同。用于指定函数的某个重载形式是不推荐的。当在程序中调用了被deprecated修饰的函数时, 编译器将给出C4996警告, 并且可以指定具体的警告信息。该警告信息可以来源于定义的宏。

例如:

```
// compile with: /W3
```

```
#define MY_TEXT "function is deprecated"
```

```
void func1(void) {}
```

```
__declspec(deprecated) void func1(int) {}
```

```
__declspec(deprecated("** this is a deprecated function **")) void  
func2(int) {}
```

```
__declspec(deprecated(MY_TEXT)) void func3(int) {}
```

```
int main() {
```

```
    func1();
```

```
    func1(1); // C4996, 警告信息: warning C4996: 'func1': was  
declared deprecated
```

```
    func2(1); // C4996, 警告信息: warning C4996: 'func2': ** this is
```

a deprecated function \*\*

```
func3(1); // C4996, 警告信息 : warning C4996: 'func3': function  
is deprecated  
}
```

\_\_declspec( dllimport , dllexport ) :

格式 :

\_\_declspec( dllimport ) declarator

\_\_declspec( dllexport ) declarator

分别用来从dll导入函数, 数据, 或对象以及从dll中导出函数, 数据, 或对象。相当于定义了dll的接口, 为它的客户exe或dll定义可使用的函数, 数据, 或对象。

将函数声明成dllexport就可以免去定义模块定义(.DEF)文件。

dllexport代替了\_\_export关键字。

被声明为dllexport的C++函数导出时的函数名将会按照C++规则经过处理。如果要求不按照C++规则进行名字处理, 请使用.def文件或使用extern "C"。

\_\_declspec( jitintrinsic ) :

格式 : \_\_declspec(jitintrinsic)

用于标记一个函数或元素是64位通用语言运行时(CLR)。主要用于Microsoft提供的某些库中。

使用jitintrinsic会在函数签名中加入MODOPT(IsJitIntrinsic)。

\_\_declspec( naked ) :

格式 : \_\_declspec(naked) declarator

此关键字仅用于x86系统, 多用于虚拟设备驱动。此关键字可以使编译器在生成代码时不包含任何注释或标记。仅可以对函数的定义使用, 不能用于数据声明、定义, 或者函数的声明。

\_\_declspec( noalias ) :

仅适用于函数, 它指出该函数是半纯粹的函数。半纯粹的函数是指仅引用或修改局部变量、参数和第一层间接参数。它是对编译器的一个承诺, 如果该函数引用全局变量或第二层间接指针参数, 则编译器会生成中断应用程序的代码。

\_\_declspec( restrict ) :

格式 : \_\_declspec(restrict) return\_type f();

仅适用于返回指针的函数声明或定义, 如, CRT的malloc函数 :

\_\_declspec(restrict) void \*malloc(size\_t size);它告诉编译器该函数返回的指针不会与任何其它的指针混淆。它为编译器提供执行编译器优化的更多信息。对于编译器来说, 最大的困难之一是确定哪些指针会与其它指针混淆, 而使用这些信息对编译器很有帮助。有必要指出, 这是对编译器的一个承诺, 编译器并不对其进行验证。如果您的程序不恰当地使用\_\_declspec(restrict), 则该程序的行为会不正确。

**noinline :**

因为在类定义中定义的成员函数默认都是inline的，\_\_declspec(naked)用于显式指定类中的某个函数不需要inline(内联)。如果一个函数很小而且对系统性能影响不大，有必要将其声明为非内联的。例如，用于处理错误情况的函数。

**noreturn :**

一个函数被\_\_declspec(noreturn)所修饰，那么它的含义是告诉编译器，这个函数不会返回，其结果是让编译器知道被修饰为\_\_declspec(noreturn)的函数之后的代码不可到达。

如果编译器发现一个函数有无返回值的代码分支，编译器将会报C4715警告，或者C2202错误信息。如果这个代码分支是因为函数不会返回从而无法到达的话，可以使用约定\_\_declspec(noreturn)来避免上述警告或者错误。

将一个期望返回的函数约定为\_\_declspec(noreturn)将导致未定义的行为。

在下面的这个例子中，main函数没有从else分支返回，所以约定函数fatal为\_\_declspec(noreturn)来避免编译或警告信息。

```
__declspec(noreturn) extern void fatal () {}  
int main() {  
    if(1)  
        return 1;  
    else if(0)  
        return 0;  
    else  
        fatal();  
}
```

**nothrow:**

格式：return-type \_\_declspec(nothrow) [call-convention] function-name ([argument-list])

可用于函数声明。告诉编译器被声明的函数以及函数内部调用的其它函数都不会抛出异常。

**novtable :**

可用于任何类声明中，但最好只用于纯接口类，即类本身从不实例化。此关键字的声明将阻止编译器对构造和析构函数的vfptr的初始化。可优化编译后代码大小。

如果试图实例化一个用\_\_declspec(novtable)声明的类然后访问类中成员，则会在运行时产生访问错误(access violation，即AV)。

**process :**

表示你的托管应用程序进程应该拥有一份指定全局变量，静态成员变量，或所有应用程序域共享的静态本地变量的拷贝。在使用/clr:pure进行编译时，应该使用\_\_declspec(process)，因为使用/clr:pure进行编译时，在默认情况下，每个应用程序域拥有一份全局和静态变量的拷贝。

在使用/c/r进行编译时，不必使用\_\_declspec(process)，因为使用/c/r进行编译时，在默认情况下，每个进程有一份全局和静态变量的拷贝。只有全局变量，静态成员变量，或本地类型的本地静态变量可以用\_\_declspec(process)修饰。

在使用/c/r:pure进行编译时，被声明为\_\_declspec(process)的变量同时也应该声明为const类型。

如果想每个应用程序域拥有一份全局变量的拷贝时，请使用appdomain。

property：

格式：

```
__declspec( property( get=get_func_name ) ) declarator  
__declspec( property( put=put_func_name ) ) declarator  
__declspec( property( get=get_func_name, put=put_func_name ) )  
declarator
```

该属性可用于类或结构定义中的非静态“虚数据成员”。实际上就是做了一个映射，把你的方法映射成属性，以供访问。get和put就是属性访问的权限，一个是读的权限，一个是写的权限。当编译器看到被property修饰的数据成员出现在成员选择符("." 或 "->")的右边的时候，它将把该操作转换成get或put方法。该修饰符也可用于类或结构定义中的空数组。

用法如下：

```
struct S {  
    int i;  
    void putprop(int j) {  
        i = j;  
    }  
    int getprop() {  
        return i;  
    }  
    __declspec(property(get = getprop, put = putprop)) int  
    the_prop;  
};
```

```
int main() {  
    S s;  
    s.the_prop = 5;  
    return s.the_prop;  
}
```

selectany：

格式：\_\_declspec(selectany) declarator

在MFC，ATL的源代码中充斥着\_\_declspec(selectany)的声明。selectany可以让我们在.h文件中初始化一个全局变量而不是只能放在.cpp中。比如有一个类，其中有一个静态变量，那么我们可以在.h中通过类似\_\_declspec(selectany) type class::variable = value;这样的代

码来初始化这个全局变量。既是该.h被多次include，链接器也会为我们剔除多重定义的错误。对于template的编程会有很多便利。

用法如下：

```
__declspec(selectany) int x1=1; //正确，x1被初始化，并且对外部可见
```

```
const __declspec(selectany) int x2 =2; //错误，在C++中，默认情况下const为static；但在C中是正确的，其默认情况下const不为static
```

```
extern const __declspec(selectany) int x3=3; //正确，x3是extern const，对外部可见
```

```
extern const int x4;  
const __declspec(selectany) int x4=4; //正确，x4是extern const，对外部可见
```

```
extern __declspec(selectany) int x5; //错误，x5未初始化，不能用__declspec(selectany)修饰
```

```
class X {  
public:  
X(int i){i++};  
int i;  
};
```

```
__declspec(selectany) X x(1); //正确，全局对象的动态初始化
```

thread：

格式：\_\_declspec(thread) declarator

声明declarator为线程局部变量并具有线程存储时限，以便链接器安排在创建线程时自动分配的存储。

线程局部存储(TLS)是一种机制，在多线程运行环境中，每个线程分配自己的局部数据。在标准多线程程序中，数据是在多个线程间共享的，而TLS是一种为每个线程分配自己局部数据的机制。

该属性只能用于数据或不含成员函数的类的声明和定义，不能用于函数的声明和定义。

该属性的使用可能会影响DLL的延迟载入。

该属性只能用于静态数据，包括全局数据对象(static和extern)，局部静态对象，类的静态数据成员；不能用于自动数据对象。

该属性必须同时用于数据的声明和定义，不管它的声明和定义是在一个文件还是多个文件。

\_\_declspec(thread)不能用作类型修饰符。

如果在类声明的同时没有定义对象，则\_\_declspec(thread)将被忽略，例如：

```
// compile with: /LD
__declspec(thread) class X
{
public:
    int l;
} x; //x是线程对象
X y; //y不是线程对象
```

下面两个例子从语义上来说是一样的：

```
__declspec(thread) class B {
public:
    int data;
} BObject; //BObject是线程对象
```

```
class B2 {
public:
    int data;
};
__declspec(thread) B2 BObject2; // BObject2是线程对象
```

uuid：

格式：\_\_declspec( uuid("ComObjectGUID") ) declarator

将具有唯一标识符号的已注册内容声明为一个变量，可使用\_\_uuidof()调用。

用法如下：

```
struct __declspec(uuid("00000000-0000-0000-
c000-000000000046")) IUnknown;
struct __declspec(uuid("{00020400-0000-0000-
c000-000000000046}")) IDispatch;
```