Visual C++ .NET. Encyklopedia

Autor: Microsoft Corporation
Tłumaczenie: Tomasz Miszkiel
ISBN: 83-7197-820-0
Tytuł oryginału: Microsoft Visual C++ .Net Language Reference
Format: B5, stron: 768

Visual C++ jest językiem programowania dostępnym w środowisku Visual Studio® .NET. Jest dynamicznym środowiskiem programowania umożliwiającym tworzenie aplikacji opartych na platformie Microsoft Windows® oraz Microsoft .NET, dynamicznych aplikacji sieci WWW oraz usług sieci WWW opartych na języku XML. „Visual C++ .NET. Encyklopedia" pomoże Ci wykorzystać rozszerzone możliwości tego języka.

Niniejsza książka jest oficjalną dokumentacją tego języka, opisującą dokładnie wszystkie jego elementy. Poruszono tutaj problemy związane głównie z programowaniem, mniejszy nacisk kładąc na czystą specyfikację języka. „Visual C++. NET. Encyklopedia" to doskonałe uzupełnienie dokumentacji dostępnej w formie elektronicznej, niezbędne dla każdego programisty .NET, korzystającego z C++.

Książka omawia:

- Słowa kluczowe.
- Modyfikatory.
- Instrukcje.
- Operatory.
- Atrybuty.
- Deklaracje.
- Przestrzenie nazw.
- Deklaratory abstrakcyjne.
- Klasy, struktury i unie.
- Rozszerzenia zarządzane.
- Opcje kompilatora.
- Opcje konsolidatora.
- Pliki wejściowe i wyjściowe programu LINK.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Dodatek B Opcje konsolidatora Visual C++ 689

2

Słowa kluczowe, modyfikatory oraz instrukcje

Słowa kluczowe od A do Z

__alignof

Zwraca wartość typu **size_t**, która jest wymaganym wyrównaniem danego typu (ang. *alignment requirement*).

```
__alignof( typ )
```

Na przykład:

Wyrażenie	Wartość
__alignof(char)	1
__alignof(short)	2
__alignof(int)	4
__alignof(__int64)	8
__alignof(float)	4
__alignof(double)	8
__alignof(char*)	4

W przypadku typów podstawowych wartość otrzymana w wyniku zastosowania operatora **__alignof** jest taka sama, jak w przypadku zastosowania operatora **sizeof**. Rozważmy jednak taki przykład:

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

W tym przypadku wartość operatora **__alignof** jest wymaganym wyrównaniem największego elementu struktury.

Podobnie będzie w przykładzie:

```
typedef __declspec(align(32)) struct { int a; } S;
// __alignof(S) wynosi 32.
```

Jednym z zastosowań operatora **__alignof** może być użycie go jako parametru niestandardowej procedury alokacji pamięci. Przykładowo, dla zdefiniowanej następującej struktury `S`, w celu alokacji pamięci na danej granicy wyrównania, można wywołać procedurę alokacji pamięci o nazwie `aligned_malloc`.

```
typedef __declspec(align(32)) struct { int a; double b; } S;
int n = 50; // rozmiar tablicy
S* p = (S*)aligned_malloc(n * sizeof(S), __alignof(S));
```

Więcej informacji na temat modyfikowania wyrównania można znaleźć w opisie:

- dyrektywy **#pragma pack**,
- atrybutu **align** specyfikatora **declspec**.

__asm

Słowo kluczowe **__asm** powoduje wywołanie wbudowanego asemblera i może występować w dowolnym miejscu, w którym dozwolona jest instrukcja języka C lub C++. Nie może występować samodzielnie. Musi po nim nastąpić instrukcja asemblera, grupa instrukcji ujęta w nawiasy klamrowe lub przynajmniej pusta para nawiasów klamrowych. Termin „blok **__asm**" odnosi się tutaj do instrukcji lub grupy instrukcji, niezależnie od tego, czy zostały one umieszczone w nawiasach klamrowych, czy też nie.

Gramatyka

instrukcja-asm:
__asm *instrukcja-asemblera* **;**$_{opc}$
__asm **{** *lista-instrukcji-asemblera* **}** **;**$_{opc}$

lista-instrukcji-asemblera:
instrukcja-asemblera **;**$_{opc}$
instrukcja-asemblera **;** *lista-instrukcji-asemblera* **;**$_{opc}$

Jeśli słowo kluczowe **__asm** zostanie użyte bez nawiasów klamrowych, oznacza ono, że pozostała część wiersza jest instrukcją asemblera. W przypadku zastosowania z nawiasami klamrowymi, słowo kluczowe **__asm** oznacza, że instrukcją asemblera jest każdy wiersz znajdujący się pomiędzy tymi nawiasami. W celu zachowania zgodności z poprzednimi wersjami, synonimem słowa kluczowego **__asm** jest słowo kluczowe **_asm**.

Z uwagi na fakt, że słowo kluczowe **__asm** jest separatorem instrukcji, instrukcje asemblera można umieszczać w tym samym wierszu.

W języku Visual C++ nie jest dostępne slowo kluczowe asm standardowego języka C++.

Przyklad

Poniższy fragment kodu jest przykładem prostego bloku **__asm** ujętego w nawiasy klamrowe:

```
__asm
{
   mov al, 2
   mov dx, 0xD007
   out dx, al
}
```

Słowo kluczowe **__asm** można również umieścić przed każdą instrukcją asemblera:

```
__asm mov al, 2
__asm mov dx, 0xD007
__asm out dx, al
```

Ponieważ słowo kluczowe **__asm** jest separatorem instrukcji, instrukcje asemblera można umieścić w tym samym wierszu:

```
__asm mov al, 2   __asm mov dx, 0xD007   __asm out dx, al
```

Dla wszystkich trzech przykładów generowany jest ten sam kod, jednak zastosowanie pierwszego stylu kodowania (tj. umieszczenia bloku **__asm** w nawiasach klamrowych) jest nieco korzystniejsze. Nawiasy wyraźnie oddzielają kod asemblera od kodu języka C lub C++, a ponadto pozwalają uniknąć zbędnego powtarzania słowa kluczowego **__asm**. Zastosowanie nawiasów klamrowych może również zapobiec niejednoznacznościom. Blok instrukcji asemblera musimy umieścić w nawiasach klamrowych, jeśli instrukcję języka C lub C++ chcemy umieścić w tym samym wierszu, w którym znajduje się blok **__asm**. Jeśli nie zastosujemy nawiasów klamrowych, kompilator nie będzie „wiedział", gdzie kończy się kod asemblera, a gdzie zaczynają się instrukcje C lub C++. Ponadto tekst w nawiasach klamrowych posiada taki sam format, jak zwykły tekst asemblera MASM, dlatego można łatwo przenieść tekst z istniejących plików źródłowych MASM.

W przeciwieństwie do nawiasów klamrowych w C oraz C++, nawiasy obejmujące blok **__asm** nie mają wpływu na zakres zmiennych. Bloki **__asm** można zagnieżdżać — zagnieżdżanie również nie wpływa na zakres zmiennych.

__assume

Wewnętrzna funkcja kompilatora **__assume** przekazuje wskazówkę do optymalizatora. Optymalizator przyjmuje, że począwszy od miejsca, gdzie występuje słowo kluczowe **__assume**, warunek reprezentowany przez *wyrażenie* jest prawdziwy i pozostaje prawdziwy do momentu modyfikacji tego *wyrażenia* (na przykład w wyniku przypisania wartości do zmiennej). Selektywne stosowanie wskazówek przekazywanych do optymalizatora za pomocą słowa kluczowego **__assume** może poprawić efekty optymalizacji.

Słowo kluczowe **__assume** należy umieszczać w obrębie makroinstrukcji `ASSERT` tylko wtedy, gdy asercja jest nienaprawialna. Funkcji **__assume** nie powinno się umieszczać w asercjach, dla których istnieje kod naprawy błędu, ponieważ w wyniku optymalizacji kod ten może zostać usunięty przez kompilator.

```
__assume(wyrażenie)
```

Funkcja **__assume** stosowana jest najczęściej w sekcji **default** instrukcji **switch**, tak jak zostało to pokazane poniżej.

Przyklad

```
// compiler_intrinsics__assume.cpp
#ifdef DEBUG
# define ASSERT(e)    ( ((e) || assert(__FILE__, __LINE__) )
#else
# define ASSERT(e)    ( __assume(e) )
#endif

void func1(int i)
{
}

void main(int p)
{
   switch(p){
      case 1:
         func1(1);
         break;
      case 2:
         func1(-1);
         break;
      default:
         __assume(0);
         // Instrukcja ta "mówi" optymalizatorowi, że przypadek domyślny
         // nie może zostać osiągnięty. Dzięki temu nie musi on generować
         // dodatkowego kodu w celu sprawdzenia, czy zmienna 'p' ma wartość,
         // która nie jest reprezentowana w gałęziach case. Dzięki temu
         // instrukcja switch działa szybciej.
   }
}
```

Uwagi

Zastosowanie instrukcji `__assume(0)` „informuje" optymalizator, że przypadek `default` nie może zostać osiągnięty. W rezultacie kompilator nie generuje kodu sprawdzającego, czy zmienna `p` ma wartość, której nie odpowiada żadna instrukcja `case`. Należy zauważyć, że aby osiągnąć taki efekt, instrukcja `__assume(0)` musi być pierwszą instrukcją w treści sekcji `default`.

Ze względu na fakt, że kompilator generuje kod w oparciu o instrukcję **__assume**, kod ten może nie działać poprawnie, jeśli podczas wykonywania wyrażenie występujące wewnątrz instrukcji **__assume** będzie miało wartość **false**. Jeśli nie ma pewności, że podczas wykonywania wyrażenie to będzie miało zawsze wartość **true**, w celu zabezpieczenia kodu można użyć makroinstrukcji **assert**:

```
# define ASSERT(e)    ( ((e) || assert(__FILE__, __LINE__)), __assume(e) )
```

Niestety, takie zastosowanie makroinstrukcji **assert** uniemożliwia kompilatorowi przeprowadzenie pokazanej wyżej optymalizacji przypadku **default**. Dlatego lepszym rozwiązaniem może być zastosowanie oddzielnej makroinstrukcji:

```
#ifdef DEBUG
# define NODEFAULT   ASSERT(0)
#else
# define NODEFAULT   __assume(0)
#endif

   default:
      NODEFAULT;
```

__based

Słowo kluczowe **__based** umożliwia deklarowanie wskaźników bazujących na innych wskaźnikach (tj. wskaźników, które są przesunięciami (ang. *offset*) od istniejących wskaźników).

typ **__based(** *baza* **)** *deklarator*

Wskaźniki oparte na adresach wskaźnikowych stanowią jedyną formę słowa kluczowego **__based**, która jest poprawna w przypadku kompilacji 32- oraz 64-bitowych. W przypadku 32-bitowego kompilatora języka C oraz C++ firmy Microsoft, tzw. wskaźnik bazujący (ang. *based pointer*) jest 32-bitowym przesunięciem od 32-bitowej bazy wskaźnika. Podobne ograniczenie występuje w przypadku środowisk 64-bitowych, gdzie wskaźnik bazujący jest 64-bitowym przesunięciem od 64-bitowej bazy.

Jednym z zastosowań wskaźników bazujących na innych wskaźnikach są tzw. identyfikatory trwałe (ang. *persistent identifiers*), które zawierają wskaźniki. Lista związana (ang. *linked list*), która składa się ze wskaźników bazujących na wskaźniku, może zostać zapisana na dysku, następnie ponownie załadowana w inne miejsce pamięci, a zawarte w niej wskaźniki nadal będą poprawne, na przykład:

```
// based_pointers1.cpp
void *vpBuffer;
struct llist_t
{
   void __based( vpBuffer ) *vpData;
   struct llist_t __based( vpBuffer ) *llNext;
};

int main()
{
}
```

Wskaźnikowi `vpBuffer` zostaje przypisany adres pamięci alokowanej później, w innym miejscu programu. Lista związana zostaje ponownie alokowana względem wartości wskaźnika `vpBuffer`.

Trwale identyfikatory zawierające wskaźniki można również zrealizować z wykorzystaniem plików odwzorowanych w pamięci (ang. *memory-mapped files*).

Gdy wykonywana jest dereferencja wskaźnika bazującego, jego baza musi być jawnie określona albo niejawnie znana przez deklarację.

W celu zachowania zgodności z poprzednimi wersjami, synonimem słowa kluczowego **__based** jest słowo kluczowe **_based**.

Przyklad

Poniższy fragment kodu demonstruje operację zmiany wartości wskaźnika bazującego przez zmianę jego bazy.

```
// based_pointers2.cpp
// skompiluj z opcją: /EHsc

#include <iostream>

int a1[] = { 1,2,3 };
int a2[] = { 10,11,12 };
int *pBased;

typedef int __based(pBased) * pBasedPtr;

using namespace std;
int main() {
   pBased = &a1[0];
   pBasedPtr pb = 0;

   cout << *pb << endl;
   cout << *(pb+1) << endl;

   pBased = &a2[0];

   cout << *pb << endl;
   cout << *(pb+1) << endl;

   return 0;
}
```

Wyjście

```
1
2
10
11
```

__cdecl

Jest to domyślna konwencja wywoływania w programach pisanych w języku C oraz C++. Ponieważ stos czyszczony jest przez kod wywołujący, konwencja ta może być stosowana dla funkcji zadeklarowanych z atrybutem **vararg** (tj. o zmiennej liczbie argumentów). W wyniku zastosowania konwencji wywoływania **__cdecl**, otrzymujemy kod wykonywalny o większym rozmiarze niż **__stdcall**, ponieważ każde wywołanie funkcji musi zawierać kod wykonujący „czyszczenie" stosu. Realizację tej konwencji wywoływania przedstawia poniższa tabela.

Element	Realizacja
Porządek przekazywania argumentów	Od strony prawej do lewej
Odpowiedzialność w zakresie obsługi stosu	Argumenty ze stosu pobiera funkcja wywołująca
Konwencja w zakresie dekoracji nazw	Nazwy poprzedzone są znakiem podkreślenia (_)
Konwencja w zakresie konwersji wielkości liter	Nie przeprowadza się konwersji wielkości liter

Modyfikator **__cdecl** należy umieścić przed nazwą zmiennej lub funkcji. Domyślnymi konwencjami nazewnictwa oraz wywoływania są konwencje języka C. W związku z tym, zastosowanie modyfikatora **__cdecl** potrzebne jest tylko wtedy, gdy ustawiona jest opcja kompilatora /Gz (**__stdcall**) lub /Gr (**__fastcall**). Opcja kompilatora /Gd wymusza konwencję wywoływania **__cdecl**.

Przyklad

W poniższym przykładzie kompilator zostaje „poinstruowany", że w przypadku funkcji `system` należy zastosować konwencję nazewnictwa i wywoływania języka C:

```
// Przykład zastosowania słowa kluczowego __cdecl w stosunku do funkcji
_CRTIMP int __cdecl system(const char *);
// Przykład zastosowania słowa kluczowego __cdecl w stosunku do wskaźnika funkcji
typedef BOOL (__cdecl *funcname_ptr)(void * arg1, const char * arg2, DWORD flags, ...);
```

__declspec

Słowo kluczowe **__declspec** jest wykorzystywane w celu określania klasy pamięci (ang. *storage-class*) przy użyciu tzw. składni atrybutów rozszerzonych (ang. *extended attribute syntax*). Za pomocą słowa kluczowego **__declspec** ustalamy, że egzemplarz danego typu ma zostać umieszczony w pamięci wraz z jednym z wymienionych poniżej atrybutów klasy pamięci specyficznych dla kompilatora Microsoftu. Przykładami innych modyfikatorów klasy pamięci są słowa kluczowe **static** oraz **extern**. Te słowa kluczowe występują jednak w specyfikacji ANSI języków C oraz C++ i dlatego nie są uwzględnione w składni atrybutów rozszerzonych. Składnia atrybutów rozszerzonych upraszcza i standaryzuje rozszerzenia wprowadzone przez firmę Microsoft do języków C oraz C++.

Oto gramatyka atrybutów rozszerzonych dla języka C++.

Gramatyka

specyfikator-deklaracji **:**
    __declspec (*sekwencja-modyfikatorów-deklaracji-rozszerzonej* **)**

*sekwencja-modyfikatorów-deklaracji-rozszerzonej***:**
    modyfikator-deklaracji-rozszerzonej$_{opc}$
    modyfikator-deklaracji-rozszerzonej sekwencja-modyfikatorów-deklaracji-rozszerzonej

modyfikator-deklaracji-rozszerzonej :
    align(#)
    allocate("*nazwasegmentu***")**
    deprecated
    dllimport
    dllexport
    naked
    noinline
    noreturn
    nothrow
    novtable
    property({get=*nazwa_funkcji_get***|, put=***nazwa_funkcji_put***})**

selectany
thread
uuid("*GUID_ObiektuCom***")**

Separatorem w sekwencji modyfikatorów deklaracji jest znak odstępu. Przykłady pojawią się w dalszej części książki.

Gramatyka atrybutów rozszerzonych udostępnia następujące atrybuty klasy pamięci specyficzne dla kompilatora Microsoftu: **align**, **allocate**, **deprecated**, **dllexport**, **dllimport**, **naked**, **noinline**, **noreturn**, **nothrow**, **novtable**, **selectany** oraz **thread**. Dostępne są również następujące atrybuty dotyczące obiektu COM: **property** oraz **uuid**.

Atrybuty klasy pamięci **thread**, **naked**, **dllexport**, **dllimport**, **nothrow**, **property**, **selectany** oraz **uuid** są właściwościami wyłącznie deklaracji obiektu lub funkcji, do której zostały zastosowane. Atrybut **thread** ma wpływ jedynie na dane i obiekty. Atrybut **naked** ma wpływ jedynie na funkcje. Atrybuty **dllimport** oraz **dllexport** wpływają na funkcje, dane oraz obiekty. Atrybuty **property**, **selectany** oraz **uuid** mają wpływ na obiekty COM.

Słowo kluczowe **__declspec** powinno zostać umieszczone na początku deklaracji prostej. Kompilator zignoruje, bez ostrzeżenia, wszystkie słowa kluczowe **__declspec** umieszczone po operatorach * lub &, a także tuż przed identyfikatorem zmiennej w deklaracji.

Atrybut **__declspec** określony na początku deklaracji typu definiowanego przez użytkownika dotyczy zmiennych tego typu, na przykład:

```
__declspec(dllimport) class X {} varX;
```

W tym przypadku atrybut dotyczy zmiennej `varX`. Atrybut **__declspec** umieszczony po słowie kluczowym **class** lub **struct** dotyczy danego typu definiowanego przez użytkownika, na przykład:

```
class __declspec(dllimport) X {};
```

W tym przypadku atrybut dotyczy typu `X`.

Ogólna zasada dotycząca stosowania atrybutu **__declspec** w przypadku deklaracji prostych jest następująca:

```
sekwencja-specyfikatorów-deklaracji lista-inicjalizowanych-deklaratorów;
```

Sekwencja-specyfikatorów-deklaracji powinna zawierać między innymi typ bazowy (np. **int**, **float**, typ **typedef** lub nazwę klasy), klasę pamięci (np. **static**, **extern**) lub rozszerzenie **__declspec**. *Lista-inicjalizowanych-deklaratorów* powinna zawierać między innymi wskaźnikową część deklaracji, na przykład:

```
__declspec(selectany) int * pi1 = 0;   //OK, zarówno atrybut "selectany", jak i typ "int"
                                       //są częścią specyfikatora-deklaracji
int __declspec(selectany) * pi2 = 0;   //OK, zarówno atrybut "selectany", jak i typ "int"
                                       //są częścią specyfikatora-deklaracji
int * __declspec(selectany) pi3 = 0;   //BŁĄD, atrybut "selectany" nie jest częścią
                                       //deklaratora
```

W poniższym fragmencie kodu zadeklarowano zmienną lokalną wątku typu **int** i przypisano jej wartość początkową.

```
// Przykład zastosowania słowa kluczowego __declspec
__declspec( thread ) int tls_i = 1;
```

__event

Deklaruje zdarzenie.

```
__event deklarator-metody;
__event __interface specyfikator-interfejsu;
__event deklarator-składowej;
```

Uwagi

Słowo kluczowe **__event** można zastosować w deklaracji metody, deklaracji interfejsu lub deklaracji danej składowej.

W zależności od tego, czy źródło oraz odbiorca zdarzeń tworzone są w rodzimym języku C++, w technologii COM, czy w tzw. kodzie zarządzanym (ang. *managed code*) platformy .NET Framework, jako zdarzeń można użyć następujących konstrukcji:

Rodzimy język C++	COM	Kod zarządzany (.NET Framework)
metoda	—	metoda
—	interfejs	—
—	—	dana składowa

W celu skojarzenia metody obsługi zdarzenia z metodą zdarzenia należy użyć instrukcji **__hook**. Należy zauważyć, że po utworzeniu zdarzenia za pomocą słowa kluczowego **__event**, w momencie wywoływania tego zdarzenia zostaną wywołane wszystkie kolejno do niego „przyczepione" procedury obsługi.

Deklaracja metody za pomocą słowa kluczowego **__event** nie może posiadać definicji — zostaje ona niejawnie wygenerowana, w wyniku czego dana metoda zdarzenia może zostać wywołana tak, jakby była zwykłą metodą.

Zdarzenia rodzime

Zdarzenia rodzime są metodami. Wartość zwracana jest zwykle wartością typu **HRESULT** lub **void**, może być jednak wartością dowolnego typu całkowitego, włącznie z typem **enum**. W przypadku, gdy zdarzenie używa wartości zwracanej typu całkowitego, warunkiem wystąpienia błędu jest zwrócenie przez metodę obsługi zdarzenia wartości niezerowej, co spowoduje, że zgłoszone zdarzenie wywoła pozostałe delegaty.

```
// Przykłady rodzimych zdarzeń języka C++:
__event void OnDblClick();
__event HRESULT OnClick(int* b, char* s);
```

Zdarzenia COM

Zdarzenia COM są interfejsami. Parametry metody w interfejsie źródła zdarzenia powinny być parametrami wejściowymi (chociaż nie jest to rygorystycznie narzucone), ponieważ parametr wyjściowy nie jest przydatny w przypadku tzw. rozsyłania grupowego (ang. *multicasting*). W przypadku zastosowania parametru wyjściowego wygenerowane zostanie ostrzeżenie poziomu 1.

Wartość zwracana jest zwykle wartością typu **HRESULT** lub **void**, może być jednak wartością dowolnego typu całkowitego, włącznie z typem wyliczeniowym. W przypadku, gdy zdarzenie używa wartości zwracanej typu całkowitego, a metoda obsługi zdarzenia zwraca wartość niezerową, dowiadujemy się, że wystąpił błąd i wywołane zdarzenie przerywa wywołania do pozostałych delegatów. Warto zauważyć, że kompilator automatycznie zaznaczy interfejs źródła zdarzenia jako źródło w wygenerowanym pliku IDL.

W przypadku źródła zdarzenia COM po słowie kluczowym **__event** należy zawsze umieścić słowo kluczowe **__interface**.

```
// Przykład zdarzenia COM:
__event __interface IEvent1;
```

Zdarzenia zarządzane

Zdarzenia zarządzane są danymi składowymi lub metodami. Typ wartości zwracanej delegatu, w przypadku gdy wykorzystywany jest on ze zdarzeniem, musi być zgodny z tzw. specyfikacją wspólnego języka CLS (*Common Language Specification*). Typ zwracany metody obsługi zdarzenia musi zgadzać się z typem zwracanym delegatu. Więcej informacji na temat delegatów można znaleźć w opisie słowa kluczowego **__delegate**. Jeśli zarządzane zdarzenie jest daną składową, to jej typ musi być wskaźnikiem do delegatu.

W środowisku .NET Framework daną składową można traktować tak, jakby sama była metodą (tj. metodą **Invoke** odpowiadającego jej delegatu). W celu zadeklarowania danej składowej zdarzenia zarządzanego należy predefiniować typ delegatu. Natomiast w przypadku metody zdarzenia zarządzanego, jeśli odpowiadający jej delegat nie został wcześniej zdefiniowany, zostaje on zdefiniowany niejawnie. Na przykład wartość zdarzeniową (ang. *event value*) taką jak `OnClick` można zdefiniować jako zdarzenie w następujący sposób:

```
// Przykłady zdarzeń zarządzanych:
__event ClickEventHandler* OnClick;  // dana składowa jako zdarzenie
__event void OnClick(String* s);     // metoda jako zdarzenie
```

W przypadku niejawnej deklaracji zdarzenia zarządzanego można zdefiniować akcesory dodające i usuwające, które będą wywoływane przy dodawaniu bądź usuwaniu procedur obsługi tego zdarzenia. Można również zdefiniować metodę, która będzie wywoływać (zgłaszać) dane zdarzenie spoza obrębu klasy.

Przyklad. Zdarzenia rodzime

```
// EventHandling_Native_Event.cpp
 [event_source(native)]
class CSource
```

```
{
public:
   __event void MyEvent(int nValue);
};

void main()
{
}
```

Przyklad. Zdarzenia COM

```
// EventHandling_COM_Event.cpp
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>

[ module(DLL, name="EventSource", uuid="6E46B59E-89C3-4c15-A6D8-B8A1CEC98830") ];

[ dual, uuid("00000000-0000-0000-0000-000000000002") ]
__interface IEventSource
{
   [id(1)] HRESULT MyEvent();
};
 [ coclass, uuid("00000000-0000-0000-0000-000000000003"), event_source(com) ]
class CSource : public IEventSource
{
public:
   __event __interface IEventSource;
   HRESULT FireEvent()
   {
      __raise MyEvent();
      return S_OK;
   }
};

void main()
{
}
```

Przyklad. Zdarzenia zarządzane

```
// EventHandling_Managed_Event.cpp
// compile with: /clr
#using <mscorlib.dll>
using namespace System;
[event_source(managed)]
public __gc class CPSource
{

public:
   __event void MyEvent(Int16 nValue);
};

void main()
{
}
```

__except

Instrukcja try-except

Gramatyka

instrukcja-try-except :
__**try** *instrukcja-złożona*
__**except** (*wyrażenie*) *instrukcja-złożona*

Instrukcja **try-except** stanowi rozszerzenie możliwości języków C oraz C++ wprowadzone przez Microsoft, które umożliwia aplikacjom przeznaczonym do środowiska 32-bitowego przejąć kontrolę w sytuacji, kiedy wystąpiły zdarzenia, które przerywają wykonywanie programu. Takie zdarzenia nazywane są wyjątkami (ang. *exception*), a mechanizm, który zajmuje się wyjątkami, zwany jest strukturalną obsługą wyjątków (ang. *structured exception handling*).

Informacje o pokrewnej tematyce można znaleźć w opisie instrukcji **try-finally**.

Wyjątki mogą być pochodzenia sprzętowego lub programowego. Nawet w sytuacji, gdy po wystąpieniu wyjątku sprzętowego lub programowego aplikacja nie może powrócić do normalnego stanu, mechanizm strukturalnej obsługi wyjątków umożliwia wyświetlenie informacji o błędzie, a także uchwycenie wewnętrznego stanu aplikacji, co może być pomocne w przeprowadzeniu diagnostyki problemu. Jest to szczególnie przydatne w przypadku problemów sporadycznych, które nie mogą zostać łatwo odtworzone.

Mechanizm strukturalnej obslugi wyjątków wspólpracuje ze środowiskiem Win32 zarówno w plikach źródlowych języka C, jak i C++. Nie zostal on jednak zaprojektowany specjalnie pod kątem języka C++. Lepszą przenośność kodu można zapewnić, stosując mechanizm obslugi wyjątków C++. Mechanizm ten jest ponadto bardziej elastyczny pod tym względem, że jest w stanie obslugiwać wyjątki dowolnego typu. W przypadku programów w języku C++ zaleca się stosować mechanizm obslugi wyjątków C++ (instrukcje try, catch oraz throw).

Instrukcja złożona występująca po klauzuli __**try** to tzw. ciało lub sekcja dozorowana (ang. *guarded section*). Instrukcja złożona występująca po klauzuli __**except** jest blokiem obsługi danego wyjątku. W bloku obsługi zdefiniowany jest zestaw operacji, które zostaną przeprowadzone w przypadku, gdy podczas wykonywania ciała sekcji dozorowanej zostanie zgłoszony wyjątek. Przebieg wykonywania jest następujący:

- Zostaje wykonana sekcja dozorowana.
- Jeśli podczas wykonywania sekcji dozorowanej nie wystąpi żaden wyjątek, wykonywanie kontynuowane jest od instrukcji występującej za klauzulą __**except**.
- Jeśli podczas wykonywania sekcji dozorowanej lub dowolnej procedury z niej wywołanej wystąpi wyjątek, obliczone zostaje wyrażenie __**except**, a jego wartość określa sposób obsługi tego wyjątku. Możliwe są trzy wartości:
  - **EXCEPTION_CONTINUE_EXECUTION (-1)** — wyjątek został odrzucony. Wykonywanie ma być kontynuowane od miejsca, w którym wystąpił wyjątek.

- **EXCEPTION_CONTINUE_SEARCH (0)** — wyjątek nie został rozpoznany. Przeszukiwanie stosu ma być kontynuowane w celu zlokalizowania bloku obsługi zawierającego instrukcje **try-except**, a następnie bloków obsługi o drugim co do wysokości priorytecie.
- **EXCEPTION_EXECUTE_HANDLER (1)** — wyjątek został rozpoznany. Sterowanie ma zostać przekazane do bloku obsługi wyjątku przez wykonanie instrukcji złożonej **__except**, a następnie wykonywanie ma być kontynuowane od instrukcji znajdującej się za blokiem **__except**.

Ze względu na to, że wyrażenie **__except** jest wyliczane jako wyrażenie języka C, jest ono ograniczone do pojedynczej wartości operatora wyrażenia warunkowego lub operatora przecinkowego. Jeśli wymagane jest bardziej rozbudowane przetwarzanie, w wyrażeniu **__except** można wywołać funkcję, która zwróci jedną z trzech wymienionych wyżej wartości.

Każda aplikacja może mieć swój własny mechanizm obsługi wyjątków. Wyrażenie **__except** wykonywane jest w zakresie ciała klauzuli **__try**. Oznacza to, że wyrażenie to ma dostęp do wszystkich zadeklarowanych w tym zakresie zmiennych lokalnych.

Skok do wnętrza instrukcji **__try** nie jest dozwolony, ale dozwolony jest skok na zewnątrz takiej instrukcji. Blok obsługi wyjątku nie zostanie wywołany, jeśli proces zostanie przerwany podczas wykonywania instrukcji **try-except**.

Funkcje wewnętrzne mechanizmu strukturalnej obslugi wyjątków

Mechanizm strukturalnej obsługi wyjątków udostępnia dwie wewnętrzne funkcje, które można zastosować wraz z instrukcją **try-except**. Funkcjami tymi są **_exception_code** oraz **_exception_info**.

Funkcja wewnętrzna **_exception_code** zwraca wartość kodu danego wyjątku (w postaci 32-bitowej liczby całkowitej).

Funkcja wewnętrzna **_exception_info** zwraca wskaźnik do struktury zawierającej dodatkowe informacje na temat tego wyjątku. Za pośrednictwem tego wskaźnika można uzyskać dostęp do takiego stanu urządzenia, jaki istniał w chwili wystąpienia wyjątku sprzętowego. Struktura ta ma następującą postać:

```
struct exception_pointers {
      EXCEPTION_RECORD *ExceptionRecord,
      CONTEXT *ContextRecord }
```

Typy wskaźnikowe **_EXCEPTION_RECORD** oraz **_CONTEXT** zdefiniowane są w pliku nagłówkowym *EXCPT.H*.

Funkcji **_exception_code** można użyć w obrębie bloku obsługi wyjątku. Funkcję **_exception_info** można jednak wykorzystać wyłącznie w obrębie filtra wyjątków. Dane, które wskazuje ta funkcja, zasadniczo znajdują się na stosie i nie będą już dostępne, gdy sterowanie zostanie przekazane do bloku obsługi wyjątku.

Funkcja wewnętrzna **_abnormal_termination** jest dostępna w obrębie bloku obsługi zakończenia. Funkcja ta zwraca wartość 0, jeśli wykonywanie ciała instrukcji **try-finally** kończy się prawidłowo. We wszystkich pozostałych przypadkach zwraca wartość 1.

Przyklad

```
// exceptions_try_except_Statement.cpp
// skompiluj z opcją: /EHsc
// Przykład zastosowania instrukcji try-except oraz try-finally
#include "stdio.h"

void main()
{
   int* p = 0x00000000;   // wskaźnik do adresu NULL
   puts("hello");
   __try{
      puts("w sekcji try");
      __try{
         puts("w sekcji try");
         *p = 13;    // powoduje wyjątek naruszenia dostępu
      }__finally{
         puts("w sekcji finally");
      }
   }__except(puts("w sekcji filtra"), 1){
      puts("w sekcji except");
   }
   puts("world");
}
```

Wyjście

```
hello
w sekcji try         // wchodzi w sekcję try
w sekcji try         // wchodzi w zagnieżdżoną sekcję try
w sekcji filtra      // wykonuje filtrowanie; zwraca wartość 1, a więc wyjątek zostaje
                     // przyjęty
w sekcji finally     // "odwija" zagnieżdżoną sekcję finally
w sekcji except      // przekazuje sterowanie do wybranego bloku obsługi
world                // opuszcza blok obsługi wyjątku
```

__fastcall

Konwencja wywoływania odpowiadająca słowu kluczowemu **__fastcall** polega na tym, że argumenty w miarę możliwości mają być przekazywane do funkcji w rejestrach. Realizację tej konwencji wywoływania przedstawia poniższa tabela.

Element	Realizacja
Porządek przekazywania argumentów	Dwa pierwsze argumenty typu DWORD lub mniejszego przekazywane są w rejestrach ECX oraz EDX; wszystkie pozostałe argumenty są przekazywane kolejno, od strony prawej do lewej.
Odpowiedzialność w zakresie obsługi stosu	Argumenty ze stosu pobiera wywoływana funkcja.
Konwencja w zakresie dekoracji nazw	Nazwy poprzedzone są znakiem @; do nazw dołączony jest również znak @, po którym następuje liczba bajtów (w notacji dziesiętnej) w liście parametrów.
Konwencja w zakresie konwersji wielkości liter	Nie przeprowadza się konwersji wielkości liter.

Przyszle wersje kompilatora do przechowywania parametrów mogą wykorzystywać inne rejestry.

Zastosowanie opcji kompilatora /Gr sprawi, że każda funkcja w danym module zostanie skompilowana jako **__fastcall**, chyba że została zadeklarowana ze sprzecznym atrybutem lub posiada nazwę `main`.

Przyklad

W poniższym przykładzie w rejestrach przekazywane są argumenty funkcji o nazwie `DeleteAggrWrapper`:

```
// Przykład zastosowania słowa kluczowego __fastcall
#define FASTCALL    __fastcall

void FASTCALL DeleteAggrWrapper(void* pWrapper);
// Przykład zastosowania słowa kluczowego __fastcall w stosunku do wskaźnika funkcji
typedef BOOL (__fastcall *funcname_ptr)(void * arg1, const char * arg2, DWORD flags, ...);
```

__finally

Instrukcja try-finally

Gramatyka

instrukcja-try-finally:
    __try *instrukcja-złożona*
    __finally *instrukcja-złożona*

Instrukcja **try**-**finally** stanowi rozszerzenie możliwości języków C oraz C++ wprowadzone przez Microsoft, które w aplikacjach przeznaczonych do środowiska 32-bitowego umożliwia zagwarantowanie wykonania kodu czyszczącego w przypadku przerwania wykonywania danego bloku kodu. Na operację czyszczenia składają się: dealokacja pamięci, zamknięcie plików oraz zwolnienie uchwytów plików. Instrukcja **try**-**finally** jest szczególnie przydatna w procedurach posiadających kilka miejsc, w których wykonywany jest test wystąpienia błędu, który mógłby spowodować przedwczesny powrót z danej procedury.

Mechanizm strukturalnej obslugi wyjątków wspólpracuje ze środowiskiem Win32 zarówno w plikach źródlowych języka C, jak i w plikach źródlowych języka C++. Nie zostal on jednak zaprojektowany specjalnie pod kątem języka C++. Lepszą przenośność kodu można zapewnić, stosując mechanizm obslugi wyjątków C++. Mechanizm ten jest ponadto bardziej elastyczny pod tym względem, że jest w stanie obslugiwać wyjątki dowolnego typu. W przypadku programów w języku C++, zaleca się stosować mechanizm obslugi wyjątków C++ (instrukcje try, catch oraz throw).

Instrukcja złożona występująca po klauzuli **__try** to tzw. sekcja dozorowana. Instrukcja złożona występująca po klauzuli **__finally** jest blokiem obsługi zakończenia. W bloku obsługi zdefiniowany jest zestaw operacji, które są wykonywane po opuszczeniu sekcji dozorowanej, niezależnie od tego, czy wykonywanie sekcji dozorowanej zakończyło się w wyniku standardowego przebiegu programu (zakończenie prawidłowe), czy też z powodu wystąpienia wyjątku (zakończenie nieprawidłowe).

Sterowanie dociera do instrukcji **__try** w wyniku zwykłego, sekwencyjnego przebiegu wykonywania, określanego angielskim terminem *fall through*. Gdy sterowanie „wchodzi" do bloku **__try**, zostaje uaktywniony skojarzony z nim blok obsługi. Jeśli nie wystąpi żaden wyjątek, przebieg wykonywania jest następujący:

- Zostaje wykonana sekcja dozorowana.
- Zostaje wywołany blok obsługi zakończenia.
- Gdy blok obsługi zakończenia kończy swoje działanie, wykonywanie kontynuowane jest od instrukcji znajdującej się za instrukcją **__finally**. Niezależnie od tego, jak kończy się wykonywanie sekcji dozorowanej (na przykład wyjściem z bloku strzeżonego za pomocą instrukcji **goto** lub instrukcją **return**), blok obsługi zakończenia wykonywany jest *zanim* sterowanie przebiegiem programu opuści sekcję dozorowaną.

  Instrukcja **__finally** nie zostanie wykonana, jeśli wyjątek zostanie przechwycony przez blok obsługi znajdujący się wyżej na stosie. Instrukcja **__finally** nie blokuje poszukiwań odpowiedniego bloku obsługi wyjątku.

Jeśli wyjątek wystąpi w bloku **__try**, kompilator musi znaleźć blok **__except**, bo inaczej wykonywanie programu zakończy się niepowodzeniem. Jeśli blok **__except** zostanie znaleziony, wykonywane są wszystkie bloki **__finally**, a następnie wykonany zostaje dany blok **__except**.

Porządek wykonywania bloku obslugi zakończenia

Slowo kluczowe __leave

Słowo kluczowe **__leave** jest dozwolone w obrębie bloku instrukcji **try-finally**. Rezultatem wykonania instrukcji **__leave** jest skok na koniec bloku **try-finally**. Wykonywanie bloku obsługi zakończenia zostaje natychmiast zakończone. Ten sam rezultat można osiągnąć za pomocą instrukcji **goto**, jednak instrukcja ta, w przeciwieństwie do bardziej wydajnej instrukcji **__leave**, powoduje „odwijanie" stosu.

Zakończenie nieprawidlowe

Za zakończenie nieprawidłowe uważane jest opuszczenie instrukcji **try-finally** za pomocą funkcji wykonawczej **longjmp**. Skok do wnętrza instrukcji **__try** nie jest dozwolony, ale dozwolony jest skok na zewnątrz takiej instrukcji. Wszystkie instrukcje, które są aktywne pomiędzy punktem wyjścia (miejscem normalnego zakończenia bloku **__try**) a punktem docelowym (blokiem **__except** obsługującym wyjątek) muszą zostać wykonane. Proces taki zwany jest „odwinięciem" lokalnym (ang. *local unwind*).

Jeśli wykonywanie bloku **__try** zostanie z jakiegoś powodu (włącznie ze skokiem poza obręb bloku) przedwcześnie zakończone, system wykona skojarzony z nim blok **__finally** jako część procesu „odwijania" stosu. W takich przypadkach funkcja **AbnormalTermination**, jeśli zostanie wywołana w obrębie bloku **finally,** zwróci wartość **true**; w przeciwnym wypadku zwróci wartość **false**.

Blok obsługi zakończenia nie zostanie wywołany, jeśli podczas wykonywania instrukcji **try-finally** proces zostanie „zabity".

__forceinline

inline, __inline, __forceinline

Specyfikatory **inline** oraz **__inline** są dla kompilatora instrukcją nakazującą wstawienie kopii treści danej funkcji we wszystkich miejscach, w których instrukcja ta jest wywoływana.

```
inline deklarator_funkcji;
__inline deklarator_funkcji;      // specyficzne dla kompilatorów Microsoftu
__forceinline deklarator_funkcji; // specyficzne dla kompilatorów Microsoftu
```

Operacja taka, zwana rozwinięciem w miejscu wywołania (ang. *inline expansion lub inlining*), następuje wyłącznie w przypadku, gdy analiza zysków i strat kompilatora wykaże, że jej wykonanie jest opłacalne. Rozwinięcie w miejscu wywołania łagodzi obciążenie związane z wywołaniem funkcji kosztem potencjalnego zwiększenia rozmiaru kodu.

Słowo kluczowe **__forceinline** unieważnia analizę zysków i strat, uzależniając decyzję o rozwinięciu w miejscu wywołania od opinii programisty. Ze specyfikatora **__forceinline** należy korzystać ostrożnie. Masowe stosowanie specyfikatora **__forceinline** może prowadzić do otrzymania kodu o dużym rozmiarze i marginalnych przyrostach wydajności lub nawet, w niektórych przypadkach, spadkach wydajności (np. z powodu zwiększonego stronicowania większego pliku wykonywalnego).

Wykorzystanie funkcji rozwijanych w miejscu wywołania może przyspieszyć wykonywanie programu, ponieważ funkcje te eliminują obciążenie związane z wywołaniami funkcji. Funkcje rozwijane w miejscu wywołania podlegają optymalizacji kodu, która jest niedostępna dla zwykłych funkcji.

Opcje oraz słowa kluczowe dotyczące rozwijania w miejscu wywołania kompilator traktuje jako sugestie. Nie ma gwarancji, że funkcje te zostaną rozwinięte w miejscu wywołania. Nie można wymusić na kompilatorze rozwinięcia w miejscu wywołania konkretnej funkcji, nawet przy użyciu słowa kluczowego **__forceinline**.

Słowo kluczowe **inline** jest dostępne wyłącznie w języku C++. Słowa kluczowe **__inline** oraz **__forceinline** są dostępne zarówno w C, jak i w C++. W celu zachowania zgodności z poprzednimi wersjami, synonimem słowa kluczowego **__inline** jest słowo kluczowe **_inline**.

Słowo kluczowe **inline** „mówi" kompilatorowi, że preferowane jest rozwinięcie w miejscu wywołania. Kompilator może jednak utworzyć oddzielne kopie danej funkcji oraz utworzyć powiązania oparte na standardowych wywołaniach, zamiast wstawiać kod w miejscu wywołania. Może się to zdarzyć w następujących przypadkach:

- funkcje rekurencyjne,
- funkcje, do których odwołujemy się za pośrednictwem wskaźnika z innego miejsca w danej jednostce translacji.

Te i inne przypadki (w zależności od uznania kompilatora) mogą kolidować z rozwinięciem w miejscu wywołania. Nie należy zatem polegać na specyfikatorze **inline**, jeśli dana funkcja ma być rozwinięta w miejscu wywołania.

Należy zauważyć, że aby funkcja mogła kandydować do rozwinięcia w miejscu wywołania, musi być zdefiniowana w nowej formie. Funkcje, które są zadeklarowane jako **inline**, a które nie są funkcjami składowymi klasy, posiadają łączność wewnętrzną (ang. *internal linkage*), chyba że określono inaczej.

Tak jak w przypadku zwykłych funkcji, nie istnieje zdefiniowany porządek wyliczania argumentów do funkcji typu *inline*. W rzeczywistości, może się on różnić do kolejności, w jakiej argumenty są wyliczane w przypadku przekazywania z wykorzystaniem protokołu wywoływania zwykłych funkcji.

Opcja optymalizacji kompilatora /Ob pomaga ustalić, czy faktycznie nastąpiło rozwinięcie funkcji w miejscu wywołania.

Przyklad 1.

```
// inline_keyword1.cpp
inline int max( int a , int b )
{
   if( a > b ) return a;
   return b;
}
void main()
{
}
```

Funkcje składowe klasy mogą zostać zadeklarowane jako *inline* albo przez zastosowanie słowa kluczowego **inline**, albo przez umieszczenie definicji funkcji wewnątrz definicji klasy.

Przyklad 2.

```
// inline_keyword2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class MyClass
{
public:
   void print() { cout << i << ' '; }   // niejawna definicja funkcji jako "inline"
private:
   int i;
};

void main()
{
}
```

Słowo kluczowe **__inline** jest równoznaczne słowu kluczowemu **inline**.

Nawet po zastosowaniu specyfikatora **__forceinline**, w pewnych okolicznościach kompilator może nie wykonać rozwinięcia funkcji w miejscu wywołania. Kompilator nie może rozwinąć funkcji w miejscu wywołania, jeżeli:

- Funkcja oraz kod, który ją wywołuje, zostały skompilowane z wykorzystaniem opcji /Ob0 (opcja domyślna w przypadku kompilacji w trybie testowania).

- Funkcja oraz kod, który ją wywołuje, wykorzystują różne typy mechanizmów obsługi wyjątków (funkcja — mechanizm obsługi wyjątków C++, kod — mechanizm strukturalnej obsługi wyjątków).
- Funkcja posiada zmienną listę argumentów.
- Funkcja korzysta z instrukcji wbudowanego asemblera, chyba że została skompilowana z wykorzystaniem opcji /Og, /Ox, /O1 lub /O2.
- Funkcja zwraca „nieodwijalny" (ang. *unwindable*) obiekt przez wartość w przypadku kompilacji z wykorzystaniem opcji /GX, /EHs lub /EHa.
- Funkcja otrzymuje „nieodwijalny" (ang. *unwindable*) obiekt utworzony w wyniku kopiowania, przekazany przez wartość w przypadku kompilacji z wykorzystaniem opcji /GX, /EHs lub /EHa.
- Funkcja jest rekurencyjna i nie towarzyszy jej dyrektywa **#pragma inline_recursion(on)**. W przypadku użycia tej dyrektywy, funkcje mogą być rozwinięte w miejscu wywołania do domyślnej głębokości ośmiu wywołań. W celu zmiany tej głębokości należy użyć dyrektywy **#pragma inline_depth**.
- Funkcja jest wirtualna i jest wirtualnie wywoływana. Wywołania bezpośrednie funkcji wirtualnych mogą zostać rozwinięte.
- Program pobiera adres funkcji i wywołanie wykonywane jest za pośrednictwem wskaźnika do tej funkcji. Bezpośrednie wywołania takich funkcji mogą zostać rozwinięte.
- Funkcja została również oznaczona modyfikatorem **__declspec(naked)**.

W przypadku, gdy kompilator nie może rozwinąć w miejscu wywołania funkcji zadeklarowanej ze specyfikatorem **__forceinline**, generowane jest ostrzeżenie poziomu 1 (o kodzie 4717).

Wywołania funkcji rekurencyjnych mogą być zastąpione ich treścią do głębokości określonej przez dyrektywę **#pragma inline_depth**. Po przekroczeniu tej głębokości, wywołania funkcji rekurencyjnej traktowane są jak wywołania kopii danej funkcji. Dyrektywa **#pragma inline_recursion** steruje rozwinięciem w miejscu wywołania funkcji, która w danej chwili poddawana jest rozwinięciu.

__hook

Kojarzy metodę obsługi ze zdarzeniem.

```
long __hook(
   &KlasaŹródła::MetodaZdarzenia,
   źródło,
   &KlasaOdbiorcy::MetodaObsługi
   [, odbiorca = this]
);
long __hook(
   interfejs,
   źródło
);
```

Parametry

&KlasaŹródła::MetodaZdarzenia

Wskaźnik do metody zdarzenia, do której „przyczepiamy" (ang. *hook*) metodę obsługi zdarzenia:

- Zdarzenia rodzime C++: *KlasaŹródła* to klasa źródła zdarzenia, a *MetodaZdarzenia* to samo zdarzenie.
- Zdarzenia COM: *KlasaŹródła* to interfejs źródła zdarzenia, a *MetodaZdarzenia* to jedna z jego metod.
- Zdarzenia zarządzane: *KlasaŹródła* to klasa źródła zdarzenia, a *MetodaZdarzenia* to samo zdarzenie.

interfejs

Nazwa interfejsu, który jest „przyczepiany" do *odbiorcy* — wyłącznie w przypadku odbiorców zdarzeń COM, w których parametr *zależny_od_układu* atrybutu **event_receiver** ma wartość **true**.

źródło

Wskaźnik do egzemplarza źródła zdarzenia. W zależności od typu kodu (parametr *typ*) określonego w atrybucie **event_receiver**, *źródło* może być:

- wskaźnikiem do obiektu źródła zdarzenia rodzimego,
- wskaźnikiem opartym na interfejsie **IUnknown** (źródło COM),
- wskaźnikiem do obiektu zarządzanego (w przypadku zdarzeń zarządzanych).

&KlasaOdbiorcy::MetodaObsługi

Wskaźnik do metody obsługi zdarzenia, która ma być „przyczepiona" do zdarzenia. Procedura obsługi jest zdefiniowana jako metoda klasy lub referencja do metody. Jeśli nazwa klasy nie zostanie określona, instrukcja **__hook** przyjmie, że jest nią nazwa klasy, w której instrukcja ta jest wywoływana.

- Zdarzenia rodzime C++: *KlasaOdbiorcy* to klasa odbiorcy zdarzenia, a *MetodaObsługi* to procedura obsługi.
- Zdarzenia COM: *KlasaOdbiorcy* to interfejs odbiorcy zdarzenia, a *MetodaObsługi* to jedna z jego procedur obsługi.
- Zdarzenia zarządzane: *KlasaOdbiorcy* to klasa odbiorcy zdarzenia, a *MetodaObsługi* to procedura obsługi.

odbiorca (parametr opcjonalny)

Wskaźnik do egzemplarza klasy odbiorcy zdarzenia. Jeśli nie określimy odbiorcy, domyślnym odbiorcą będzie klasa lub struktura odbiorcy, wewnątrz której została wywołana funkcja **__hook**.

Stosowanie

Funkcja może być używana lokalnie (wyłącznie w treści metody klasy lub struktury odbiorcy zdarzenia).

Uwagi

Funkcję wewnętrzną **__hook** należy stosować wewnątrz odbiorcy zdarzenia w celu skojarzenia lub „przyczepienia" metody obsługi do metody zdarzenia. Określona w ten sposób procedura obsługi zostanie wywołana, gdy źródło zgłosi podane zdarzenie. Do pojedynczego zdarzenia można „przyczepić" kilka procedur obsługi, a do pojedynczej procedury obsługi można „przyczepić" kilka zdarzeń.

Istnieją dwie postacie funkcji **__hook**. W większości przypadków można stosować pierwszą z jej postaci (czteroargumentową), szczególnie w przypadku odbiorców zdarzeń COM, których parametr *zależny_od_układu* atrybutu **event_receiver** ma wartość **false**.

W takich przypadkach nie trzeba „przyczepiać" wszystkich metod w interfejsie przed „odpaleniem" zdarzenia — „przyczepiona" musi być jedynie metoda obsługująca dane zdarzenie. Drugą (dwuargumentową) postać funkcji **__hook** można zastosować jedynie w przypadku odbiorcy zdarzenia COM, którego parametr *zależny_od_układu* ma wartość **true**.

Funkcja **__hook** zwraca wartość typu **long**. Niezerowa wartość zwracana wskazuje na wystąpienie błędu (zdarzenia zarządzane zgłoszą w takiej sytuacji wyjątek).

Kompilator sprawdza, czy zdarzenie istnieje oraz czy sygnatura zdarzenia jest zgodna z sygnaturą delegatu.

Z wyjątkiem przypadku, kiedy mamy do czynienia ze zdarzeniami COM, funkcje **__hook** oraz **__unhook** mogą zostać wywołane poza odbiorcą zdarzenia.

__identifier

Umożliwia wykorzystanie słów kluczowych języka C++ jako identyfikatorów.

```
__identifier(słowo_kluczowe_C++)
```

Uwagi

Słowo kluczowe **__identifier** umożliwia wykorzystanie słów kluczowych języka C++ jako identyfikatorów. Głównym zastosowaniem tego słowa kluczowego jest przekazanie klasom zarządzanym dostępu do klas zewnętrznych i umożliwienie korzystania z nich. W klasach zewnętrznych rolę identyfikatorów mogą pełnić słowa kluczowe języka C++.

Zastosowanie slowa kluczowego __identifier w stosunku do identyfikatorów, które nie są slowami kluczowymi jest wprawdzie dozwolone, lecz ze względów stylistycznych zdecydowanie odradzane.

Przyklad

W poniższym przykładzie zostaje utworzona klasa języka C# (o nazwie `template`), która następnie zostaje przypisana wskaźnikowi `pTemplate`.

```
// identifier_template.cs
// skompiluj z opcją: /target:library
public class template
```

```
{
   public void Run()
   {
   }
}
// keyword__identifier.cpp
// skompiluj z opcją: /clr
#using <mscorlib.dll>
#using <identifier_template.dll>   // definiuje klasę języka C# o nazwie template

void main()
{
   __identifier(template) *pTemplate = new __identifier(template)();
   pTemplate->Run();
}
```

__if_exists

Słowo kluczowe **__if_exists** pozwala na warunkowe dołączanie kodu. Dołączenie nastąpi tylko wtedy, gdy zostanie stwierdzone istnienie określonego symbolu.

```
__if_exists ( zmienna ) {
instrukcje
}
```

gdzie:

zmienna

Symbol, którego istnienie chcemy sprawdzić.

instrukcje

Jedna lub więcej instrukcji, które mają zostać wykonane, jeśli *zmienna* istnieje.

Uwagi

Słowo kluczowe **__if_exists** może być wykorzystywane podczas weryfikacji faktu istnienia zarówno identyfikatorów składowych, jak i identyfikatorów nie będących składowymi. Ponadto za pomocą **__if_exists** można zweryfikować fakt istnienia funkcji przeciążonej, nie można jednak sprawdzić, czy istnieje konkretna forma przeciążenia.

Słowo kluczowe **__if_not_exists** pozwala na warunkowe dołączanie kodu. Dołączenie nastąpi tylko wtedy, gdy zostanie spełniony warunek, że określony symbol nie istnieje.

Przyklad

```
// the__if_exists_statement.cpp
// skompiluj z opcją: /EHsc
#include <iostream>

template<typename T>
class X : public T {
public:
   void Dump() {
      std::cout << "Wewnątrz funkcji X<T>::Dump()" << std::endl;
```

```
      __if_exists(T::Dump) {
         T::Dump();
      }

      __if_not_exists(T::Dump) {
         std::cout << "Funkcja T::Dump nie istnieje" << std::endl;
      }
   }
};

class A {
public:
   void Dump() {
      std::cout << "Wewnątrz funkcji A::Dump()" << std::endl;
   }
};

class B {
};

bool g_bFlag = true;

class C {
public:
   void f(int);
   void f(double);
};

int main()
{
   X<A> x1;
   X<B> x2;

   x1.Dump();
   x2.Dump();

   __if_exists(::g_bFlag) {
      std::cout << "g_bFlag = " << g_bFlag << std::endl;
   }

   __if_exists(C::f) {
      std::cout << "Funkcja C::f istnieje" << std::endl;
   }

   return 0;
}
```

Wyjście

```
Wewnątrz funkcji X<T>::Dump()
Wewnątrz funkcji A::Dump()
Wewnątrz funkcji X<T>::Dump()
Funkcja T::Dump nie istnieje
g_bFlag = 1
Funkcja C::f istnieje
```

__if_not_exists

Słowo kluczowe **__if_not_exists** pozwala na warunkowe dołączanie kodu. Dołączenie nastąpi tylko wtedy, gdy zostanie spełniony warunek, że określony symbol nie istnieje.

```
__if_not_exists ( zmienna ) {
instrukcje
}
```

gdzie:

zmienna

Symbol, którego istnienie chcemy sprawdzić.

instrukcje

Jedna lub więcej instrukcji, które mają zostać wykonane, jeśli *zmienna* nie istnieje.

Uwagi

Instrukcję **__if_not_exists** można zastosować do identyfikatorów występujących zarówno wewnątrz, jak i na zewnątrz klasy. W przypadku weryfikacji faktu istnienia funkcji przeciążonych, nie można sprawdzać, czy istnieje konkretna postać przeciążenia.

Słowo kluczowe **__if_exists** pozwala na warunkowe dołączanie kodu, w zależności od faktu istnienia określonego symbolu.

Przyklad

Przykład pokazujący sposób wykorzystania instrukcji **__if_not_exists** można znaleźć w punkcie poświęconym słowu kluczowemu **__if_exists**.

__inline

inline, __inline, __forceinline

Użycie specyfikatorów **inline** oraz **__inline** jest dla kompilatora instrukcją nakazującą wstawienie kopii treści danej funkcji we wszystkich miejscach, w których instrukcja ta jest wywoływana.

```
inline deklarator_funkcji;
__inline deklarator_funkcji;      // specyficzne dla kompilatorów Microsoftu
__forceinline deklarator_funkcji; // specyficzne dla kompilatorów Microsoftu
```

Operacja taka, zwana rozwinięciem w miejscu wywołania (ang. *inline expansion lub inlining*), następuje wyłącznie w przypadku, gdy analiza zysków i strat kompilatora wykaże, że jej wykonanie jest opłacalne. Rozwinięcie w miejscu wywołania łagodzi obciążenie związane z wywołaniem funkcji kosztem potencjalnego zwiększenia rozmiaru kodu.

Słowo kluczowe **__forceinline** unieważnia analizę zysków i strat, uzależniając decyzję o rozwinięciu w miejscu wywołania od opinii programisty. Ze specyfikatora **__forceinline** należy korzystać ostrożnie. Masowe stosowanie specyfikatora **__forceinline** może prowadzić

do otrzymania kodu o dużym rozmiarze i marginalnych przyrostach wydajności lub nawet, w niektórych przypadkach, spadkach wydajności (np. z powodu zwiększonego stronicowania większego pliku wykonywalnego).

Wykorzystanie funkcji rozwijanych w miejscu wywołania może przyspieszyć wykonywanie programu, ponieważ funkcje te eliminują obciążenie związane z wywołaniami funkcji. Funkcje rozwijane w miejscu wywołania podlegają optymalizacji kodu, która jest niedostępna dla zwykłych funkcji.

Opcje oraz słowa kluczowe dotyczące rozwijania w miejscu wywołania kompilator traktuje jako sugestie. Nie ma gwarancji, że funkcje te zostaną rozwinięte w miejscu wywołania. Nie można wymusić na kompilatorze rozwinięcia w miejscu wywołania konkretnej funkcji, nawet przy użyciu słowa kluczowego **__forceinline**.

Słowo kluczowe **inline** jest dostępne wyłącznie w języku C++. Słowa kluczowe **__inline** oraz **__forceinline** są dostępne zarówno w C, jak i w C++. W celu zachowania zgodności z poprzednimi wersjami, synonimem słowa kluczowego **__inline** jest słowo kluczowe **_inline**.

Słowo kluczowe **inline** „mówi" kompilatorowi, że preferowane jest rozwinięcie w miejscu wywołania. Kompilator może jednak utworzyć oddzielne kopie danej funkcji oraz utworzyć powiązania oparte na standardowych wywołaniach, zamiast wstawiać kod w miejscu wywołania. Może się to zdarzyć w następujących przypadkach:

- funkcje rekurencyjne,
- funkcje, do których odwołujemy się za pośrednictwem wskaźnika z innego miejsca w danej jednostce translacji.

Te i inne przypadki (w zależności od uznania kompilatora) mogą kolidować z rozwinięciem w miejscu wywołania. Nie należy zatem polegać na specyfikatorze **inline**, jeśli dana funkcja ma być rozwinięta w miejscu wywołania.

Należy zauważyć, że aby funkcja mogła kandydować do rozwinięcia w miejscu wywołania, musi być zdefiniowana w nowej formie. Funkcje, które są zadeklarowane jako **inline**, a które nie są funkcjami składowymi klasy, posiadają łączność wewnętrzną (ang. *internal linkage*), chyba że zostało to określone inaczej.

Tak jak w przypadku zwykłych funkcji, nie istnieje zdefiniowany porządek wyliczania argumentów do funkcji typu *inline*. W rzeczywistości może się on różnić do kolejności, w jakiej argumenty są wyliczane w przypadku przekazywania z wykorzystaniem protokołu wywoływania zwykłych funkcji.

Opcja optymalizacji kompilatora /Ob pomaga ustalić, czy faktycznie nastąpiło rozwinięcie funkcji w miejscu wywołania.

Przyklad 1.

```
// inline_keyword1.cpp
inline int max( int a , int b )
{
   if( a > b ) return a;
   return b;
}
```

```
void main()
{
}
```

Funkcje składowe klasy mogą zostać zadeklarowane jako *inline* albo przez zastosowanie słowa kluczowego **inline**, albo przez umieszczenie definicji funkcji wewnątrz definicji klasy.

Przyklad 2.

```
// inline_keyword2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;


class MyClass
{
public:
   void print() { cout << i << ' '; }   // niejawna definicja funkcji jako "inline"
private:
   int i;
};

void main()
{
}
```

Słowo kluczowe **__inline** jest równoznaczne słowu kluczowemu **inline**.

Nawet po zastosowaniu specyfikatora **__forceinline**, w pewnych okolicznościach kompilator może nie wykonać rozwinięcia funkcji w miejscu wywołania. Kompilator nie może rozwinąć funkcji w miejscu wywołania, jeżeli:

- Funkcja oraz kod, który ją wywołuje, zostały skompilowane za pomocą opcji /Ob0 (opcja domyślna w przypadku kompilacji w trybie testowania).
- Funkcja oraz kod, który ją wywołuje wykorzystują różne typy mechanizmów obsługi wyjątków (funkcja — mechanizm obsługi wyjątków C++ , kod — mechanizm strukturalnej obsługi wyjątków).
- Funkcja posiada zmienną listę argumentów.
- Funkcja korzysta z instrukcji wbudowanego asemblera, chyba że została skompilowana za pomocą opcji /Og, /Ox, /O1 lub /O2.
- Funkcja zwraca „nieodwijalny" (ang. *unwindable*) obiekt przez wartość w przypadku kompilacji z wykorzystaniem opcji /GX, /EHs lub /EHa.
- Funkcja otrzymuje „nieodwijalny" (ang. *unwindable*) obiekt utworzony w wyniku kopiowania, przekazany przez wartość, w przypadku kompilacji z wykorzystaniem opcji /GX, /EHs lub /EHa.
- Funkcja jest rekurencyjna i nie towarzyszy jej dyrektywa **#pragma inline_recursion(on)**. W przypadku użycia tej dyrektywy, funkcje mogą zostać rozwinięte w miejscu wywołania do domyślnej głębokości ośmiu wywołań. W celu zmiany tej głębokości należy użyć dyrektywy **#pragma inline_depth**.

- Funkcja jest wirtualna i jest wirtualnie wywoływana. Wywołania bezpośrednie funkcji wirtualnych mogą zostać rozwinięte.
- Program pobiera adres funkcji i wywołanie wykonywane jest za pośrednictwem wskaźnika do tej funkcji. Bezpośrednie wywołania takich funkcji mogą zostać rozwinięte.
- Funkcja została również oznaczona modyfikatorem **__declspec(naked)**.

W przypadku, gdy kompilator nie może rozwinąć w miejscu wywołania funkcji zadeklarowanej z wykorzystaniem specyfikatora **__forceinline**, generowane jest ostrzeżenie poziomu 1 (o kodzie 4717).

Wywołania funkcji rekurencyjnych mogą zostać zastąpione ich treścią do głębokości określonej przez dyrektywę **#pragma inline_depth**. Po przekroczeniu tej głębokości, wywołania funkcji rekurencyjnej traktowane są jak wywołania kopii danej funkcji. Dyrektywa **#pragma inline_recursion** steruje rozwinięciem w miejscu wywołania funkcji, która w danej chwili poddawana jest rozwinięciu. Informacje o pokrewnej tematyce można znaleźć w opisie opcji kompilatora *Inline-Function Expansion* (/Ob) (rozwijanie funkcji w miejscu wywołania).

__int8, __int16, __int32, __int64

Język Microsoft C/C++ udostępnia obsługę typów całkowitych o określonym rozmiarze (ang. *sized integer types*). Za pomocą specyfikatora typu **__int***n*, gdzie *n* to 8, 16, 32 lub 64, można zadeklarować 8-, 16-, 32- lub 64-bitowe zmienne całkowite.

Poniższy fragment kodu zawiera przykład deklaracji jednej zmiennej dla każdego typu całkowitego o określonym rozmiarze:

```
__int8 nSmall;      // deklaracja 8-bitowej zmiennej całkowitej
__int16 nMedium;    // deklaracja 16-bitowej zmiennej całkowitej
__int32 nLarge;     // deklaracja 32-bitowej zmiennej całkowitej
__int64 nHuge;      // deklaracja 64-bitowej zmiennej całkowitej
```

Typy **__int8**, **__int16** oraz **__int32** są synonimami typów ANSI o tym samym rozmiarze i są przydatne podczas tworzenia kodu przenośnego, który zachowuje się w ten sam sposób na wielu platformach. Typ danych **__int8** jest równoważnikiem typu **char**, typ **__int16** jest równoważnikiem typu **short**, a typ **__int32** jest równoważnikiem typu **int**. Typ **__int64** nie ma odpowiednika w standardzie ANSI.

Przyklad

Poniższy przykład pokazuje, że parametr typu **__int***xx* zostanie promowany do typu **int**:

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

void func(int i) {
   printf("%s\n", __FUNCTION__);
}
```

```
void main() {
   __int8 i8 = 100;
   func(i8);   // nie ma funkcji void func(__int8 i8)
               // zmienna typu __int8 zostanie promowana do typu int
}
```

Wyjście

```
func
```

__interface

Interfejs w języku Visual C++ może być zdefiniowany następująco:

- może dziedziczyć z dowolnej liczby interfejsów bazowych lub nie dziedziczyć z żadnego,
- nie może dziedziczyć z klasy bazowej,
- może zawierać wyłącznie publiczne, czysto wirtualne metody,
- nie może zawierać konstruktorów, destruktorów i operatorów,
- nie może zawierać metod statycznych,
- nie może zawierać danych składowych, dozwolone są natomiast właściwości,

modyfikator **__interface** *nazwa-interfejsu* {*definicja-interfejsu*}

Zastosowanie słowa kluczowego **__interface** narzuca przestrzeganie tych reguł, chociaż reguły te można również zastosować do implementacji klasy lub struktury języka C++.

Przykładową definicję interfejsu zawiera następujący fragment kodu:

```
__interface IMyInterface
{
   HRESULT CommitX();
   HRESULT get_X(BSTR* pbstrName);
};
```

Warto zauważyć, że nie trzeba jawnie wskazywać, iż funkcje `CommitX` oraz `get_X` są czysto wirtualne. Równoważna deklaracja w przypadku pierwszej funkcji miałaby postać:

```
virtual HRESULT CommitX() = 0;
```

Istnieje możliwość zdefiniowania interfejsu zarządzanego, na przykład:

```
__gc __interface IMyInterface
{
};
```

Interfejsy zarządzane można implementować wyłącznie w przypadku klas zarządzanych. Więcej informacji można znaleźć w opisie słowa kluczowego **__gc**.

Słowo kluczowe **__interface** implikuje użycie modyfikatora **__declspec(novtable)**.

Przyklad

Poniższy przykład pokazuje, jak korzystać z właściwości zadeklarowanych w interfejsie.

```
// deriv_interface.cpp
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>
#include <string.h>
#include <comdef.h>
#include <stdio.h>

[module(name="test")];

[ object, uuid("00000000-0000-0000-0000-000000000001"), library_block ]
__interface IFace {
   [ id(0) ] int int_data;
   [ id(5) ] BSTR bstr_data;
};

[ coclass, uuid("00000000-0000-0000-0000-000000000002") ]
class MyClass : public IFace {
private:
   int m_i;
   BSTR m_bstr;

public:
   MyClass() {
      m_i = 0;
      m_bstr = 0;
   }

   ~MyClass() {
      if (m_bstr)
         ::SysFreeString(m_bstr);
   }

   int get_int_data() {
      return m_i;
   }

   void put_int_data(int _i) {
      m_i = _i;
   }

   BSTR get_bstr_data() {
      BSTR bstr = ::SysAllocString(m_bstr);
      return bstr;
   }

   void put_bstr_data(BSTR bstr) {
      if (m_bstr)
         ::SysFreeString(m_bstr);
      m_bstr = ::SysAllocString(bstr);
   }
};

int main() {
   _bstr_t bstr("Testowanie");
```

```
    CoInitialize(NULL);
    CComObject<MyClass>* p;
    CComObject<MyClass>::CreateInstance(&p);
    p->int_data = 100;
    printf("p->int_data = %d\n", p->int_data);
    p->bstr_data = bstr;
    printf("bstr_data = %S\n", p->bstr_data);
}
```

Wyjście

```
p->int_data = 100
bstr_data = Testowanie
```

__leave

Instrukcja try-finally

Gramatyka

instrukcja-try-finally:
    __try *instrukcja-złożona*
    __finally *instrukcja-złożona*

Slowo kluczowe __leave

Słowo kluczowe **__leave** jest dozwolone w obrębie bloku instrukcji **try**-**finally**. Rezultatem wykonania instrukcji **__leave** jest skok na koniec bloku **try**-**finally**. Wykonywanie bloku obsługi zakończenia zostaje natychmiast zakończone. Ten sam rezultat można osiągnąć za pomocą instrukcji **goto**, jednak instrukcja ta, w przeciwieństwie do bardziej wydajnej instrukcji **__leave**, powoduje „odwijanie" stosu.

Instrukcja **try**-**finally** stanowi rozszerzenie możliwości języków C oraz C++ wprowadzone przez Microsoft, które w przypadku aplikacji przeznaczonych do środowiska 32-bitowego umożliwia wykonanie kodu czyszczącego w przypadku przerwania wykonywania danego bloku kodu. Operacja czyszczenia składa się z dealokacji pamięci, zamknięcia plików i zwolnienia uchwytów plików. Instrukcja **try**-**finally** jest szczególnie przydatna w przypadku procedur posiadających kilka miejsc, w których wykonywany jest test na wystąpienie błędu, który mógłby spowodować przedwczesny powrót z danej procedury.

Informacje o pokrewnej tematyce oraz przykłady kodu można znaleźć w punkcie poświęconym instrukcji **try**-**except**.

Mechanizm strukturalnej obslugi wyjątków wspólpracuje ze środowiskiem Win32 zarówno w plikach źródlowych języka C, jak i w plikach źródlowych języka C++. Nie zostal on jednak zaprojektowany specjalnie pod kątem języka C++. Lepszą przenośność kodu można zapewnić, stosując mechanizm obslugi wyjątków C++. Mechanizm ten jest ponadto bardziej elastyczny pod tym względem, że jest w stanie obslugiwać wyjątki dowolnego typu. W przypadku programów w języku C++, zaleca się stosować mechanizm obslugi wyjątków C++ (instrukcje try, catch oraz throw).

Instrukcja złożona występująca po klauzuli **__try** to tzw. sekcja dozorowana. Instrukcja złożona występująca po klauzuli **__finally** jest blokiem obsługi zakończenia. W bloku obsługi zdefiniowany jest zestaw operacji, które są wykonywane po opuszczeniu sekcji dozorowanej, niezależnie od tego, czy wykonywanie sekcji dozorowanej zakończyło się w wyniku standardowego przebiegu programu (zakończenie prawidłowe), czy też z powodu wystąpienia wyjątku (zakończenie nieprawidłowe).

Sterowanie dociera do instrukcji **__try** w wyniku zwykłego, sekwencyjnego przebiegu wykonywania, określanego angielskim terminem *fall through*. Gdy sterowanie „wchodzi" do bloku **__try**, uaktywniony zostaje skojarzony z nim blok obsługi. Jeśli nie wystąpi żaden wyjątek, przebieg wykonywania jest następujący:

- Zostaje wykonana sekcja dozorowana.
- Zostaje wywołany blok obsługi zakończenia.
- Gdy blok obsługi zakończenia kończy swoje działanie, wykonywanie kontynuowane jest od instrukcji znajdującej się za instrukcją **__finally**. Niezależnie od tego, jak kończy się wykonywanie sekcji dozorowanej (na przykład wyjściem z bloku strzeżonego za pomocą instrukcji **goto** lub instrukcją **return**), blok obsługi zakończenia wykonywany jest, *zanim* sterowanie przebiegiem programu opuści sekcję dozorowaną.

  Instrukcja **__finally** nie zostanie wykonana, jeśli wyjątek zostanie przechwycony przez blok obsługi znajdujący się wyżej na stosie. Instrukcja **__finally** nie blokuje poszukiwań odpowiedniego bloku obsługi wyjątku.

Jeśli wyjątek wystąpi w bloku **__try**, kompilator musi znaleźć blok **__except**, bo inaczej wykonywanie programu zakończy się niepowodzeniem. Jeśli blok **__except** zostanie znaleziony, wykonywane są wszystkie bloki **__finally**, a następnie wykonany zostaje dany blok **__except**.

Porządek wykonywania bloku obslugi zakończenia

Zakończenie nieprawidlowe

Za zakończenie nieprawidłowe uważane jest opuszczenie instrukcji **try**-**finally** za pomocą funkcji wykonawczej **longjmp**. Skok do wnętrza instrukcji **__try** nie jest dozwolony, ale dozwolony jest skok na zewnątrz takiej instrukcji. Wszystkie instrukcje, które są aktywne pomiędzy punktem wyjścia (miejscem normalnego zakończenia bloku **__try**) a punktem docelowym (blok **__except** obsługujący wyjątek), muszą zostać wykonane. Proces taki zwany jest lokalnym „odwinięciem" (ang. *local unwind*).

Jeśli wykonywanie bloku **__try** zostanie z jakiegoś powodu (włącznie ze skokiem poza obręb bloku) przedwcześnie zakończone, system wykona skojarzony z nim blok **__finally** jako część procesu „odwijania" stosu. W takich przypadkach funkcja **AbnormalTermination**, jeśli zostanie wywołana w obrębie bloku **finally**, zwróci wartość **true**, w przeciwnym wypadku zwróci wartość **false**.

Blok obsługi zakończenia nie zostanie wywołany, jeśli podczas wykonywania instrukcji **try**-**finally** proces zostanie „zabity".

__m64

Typ danych **__m64**, przeznaczony do stosowania z instrukcjami wewnętrznymi MMX oraz 3DNow!, zdefiniowany jest następująco:

```
// data_types__m64.cpp
typedef union __declspec(intrin_type) __declspec(align(8)) __m64
{
   unsigned __int64 m64_u64;
   float m64_f32[2];
   __int8 m64_i8[8];
   __int16 m64_i16[4];
   __int32 m64_i32[2];
   __int64 m64_i64;
   unsigned __int8 m64_u8[8];
   unsigned __int16 m64_u16[4];
   unsigned __int32 m64_u32[2];
} __m64;
int main()
{
}
```

Do pól wchodzących w skład typu **__m64** nie powinno się odwoływać bezpośrednio. Typy te są jednak widoczne w oknie programu uruchomieniowego. Zmienna typu **__m64** zostaje odwzorowana w rejestrach MM[0-7].

Zmienne typu **__m64** są automatycznie wyrównywane do granicy 8 bajtów.

__m128

Typ danych **__m128**, przeznaczony do stosowania z instrukcjami wewnętrznymi technologii Streaming SIMD Extension oraz Streaming SIMD Extension 2, zdefiniowany jest następująco:

```
// data_types__m128.cpp
typedef struct __declspec(intrin_type) __declspec(align(16)) __m128 {
   float m128_f32[4];
} __m128;

int main()
{
}
```

Do pól wchodzących w skład typu **__m128** nie powinno się odwoływać bezpośrednio. Typy te widoczne są jednak w oknie programu uruchomieniowego. Zmienna typu **__m128** zostaje odwzorowana w rejestrach XMM[0-7].

Zmienne typu **__m128** są automatycznie wyrównywane do granicy 16 bajtów.