

Języki Programowania na Platformie .NET

Visual C++

Rodzaje kodu

- mieszany (*mixed*), /clr: moduł zawiera zarówno kod zarządzany jak i niezarządzany.
- czysty (*pure*), /clr:pure: posiada zarówno natywne i zarządzane typy danych, lecz wyłącznie zarządzane funkcje
- bezpieczny (*safe*), /clr:safe: tylko zarządzany kod

Zalety /clr:pure

- większa wydajność,
- kod istnieje wewnątrz AppDomain
- brak konieczności umieszczania modułów na dysku w celu kompilacji i uruchomienia
- pełne wsparcie dla *reflection*

Jedna z wad: brak możliwości wywoływania takiego kodu przez kod niezarządzany.

Zalety /clr:safe

- większe bezpieczeństwo
- w niektórych sytuacjach wymagane
- nieuchronnie rosnąca popularność w przyszłości

Definicja klasy – przykład

```
ref class A {  
    public:  
        void f() {  
  
        }  
};
```

```
A &a;  
a->f();  
a = nullptr;
```

Definicja klasy

```
class_access ref class name modifier : inherit_access base_type {};  
class_access ref struct name modifier : inherit_access base_type {};  
class_access value class name modifier : inherit_access base_type {};  
class_access value struct name modifier : inherit_access base_type {};
```

- `ref` informuje kompilator, że klasa lub struktura będzie umieszczona na stercku i referencja do niej będzie używana jako parametr funkcji lub składowa w klasach,
- `value` informuje kompilator, że zawartość klasy będzie przekazywana do funkcji/przechowywana w zmiennych.

Definicja klasy c.d.

- `class_access` to `public` lub `private`
- `inherit_access`: `public`
- `modifier` to `abstract` lub `sealed`
- składowe klasy są domyślnie prywatne, natomiast składowe struktury są domyślnie publiczne
- typ `'value'` nie może być typem bazowym
- funkcje wirtualne w typach referencyjnych muszą być deklarowane w klasach pochodnych z użyciem słowa kluczowego `override` lub `new` (przykład 1)
- do dynamicznego tworzenia klas używamy `gcnew`

Zmienne na stosie a GC

- gdy deklarujemy zmienną 'na stosie' (*stacs semantics*), kompilator wygeneruje kod tworzący obiekt na sterckie za pomocą `gcnew`
- deklaracja argumentu do funkcji lub zwracanej wartości z użyciem *stacs semantics* sprawia, że funkcja nie może być używana przez inne języki
- kompilator nie generuje konstruktorów kopiujących dla typów referencyjnych, należy więc zdefiniować samodzielnie `(C(C%) {})`
- kompilator również nie generuje domyślnego operatora przypisania, więc możliwość tworzenia obiektów z użyciem *stacs semantics* wymaga jego definicji `(void operator=(C%) {})`

Zmienne na stosie a GC cd

- kompilator dostarcza jednoargumentowego operatora % konwertującego zmienną zadeklarowaną z użyciem *stack semantics* do odpowiadającego typu referencyjnego
- nie można powyższej notacji używać w przypadku typów `delegate`, `array`, `String`

Zmienne na stosie a GC — przykład

```
ref class R {  
public:  
    int i;  
    R(){}  
  
    void operator=(R% r) { // operator przypisania  
        i = r.i;  
    }  
  
    R(R% r) : i(r.i) {} // konstruktor kopiujący  
};  
  
void Test(R r) {} // wymaga konstruktora kopiującego
```

Zmienne na stosie a GC — przykład cd

```
R r1; r1.i = 98;
```

```
R r2(r1); // wymaga konstruktora kopiującego  
System::Console::WriteLine(r1.i); // 98  
System::Console::WriteLine(r2.i); // 98
```

```
R ^ r3 = %r1; // % konwertuje obiekt 'ze stosu' do typu referencyjnego  
System::Console::WriteLine(r3->i); // 98  
Test(r1);
```

```
R r4, r5;  
r5.i = 13;  
r4 = r5; // wymaga zdefiniowania operatora przypisania  
System::Console::WriteLine(r4.i); // 13
```

```
R % r6 = r4;  
System::Console::WriteLine(r6.i); // 13
```

Destruktory

```
class A {  
    ~A() {} // destruktork  
    !A() {} // finalizer  
};
```

- druga z metod jest destruktorem wywoływanym przy pracy GC
- niebezpieczne jest odwoływanie się do zarządzanych danych w metodzie !A (mogły już zostać zwolnione, ponieważ *garbage collector* zwalnia zarządzane zasoby niedeterministycznie),
- metoda !A nie jest równoważna metodzie Finalize (automatyczne wywoływanie w całej hierarchii; dokumentacja clr)
- GC wywołuje Finalize, która wywołuje wiele destruktorków poprzedzonych '!', znajdujących się w hierarchii dziedziczenia
- destruktork wywołując !A nie powoduje ciągu wywołań w klasach bazowych

Typowy schemat destruktor-finalizer

```
ref class A {  
    // destruktor (zwalnia wszystkie zasoby)  
    ~A() {  
        // zwolnij zarządzane zasoby  
        // następnie wywołaj finalizer:  
        this->!A();  
    }  
  
    // finalizer (zwalnia niezarządzane zasoby)  
    !A() {  
        // zwolnij niezarządzane zasoby  
    }  
};
```

Kiedy wywoływany jest destruktor?

- jeśli obiekt został utworzony za pomocą *stack semantics* i sterowanie opuściło dany zakres,
- gdy pojawi się wyjątek podczas tworzenia obiektu,
- jeśli obiekt jest składową, której destruktor został uruchomiony,
- gdy zostanie użyty operator `delete` na referencji do obiektu,
- gdy jawnie wywołamy destruktor.

Referencja %

```
ref struct A {
    void Test(unsigned int &){}
    void Test2(unsigned int %){}
    unsigned int i;
};

int main() {
    A a;
    a.i = 9;
    a.Test(a.i);    // C2664
    a.Test2(a.i);  // OK
    unsigned int j = 0;
    a.Test(j);     // OK
}
```

Uchwyt

- wskaźniki * i referencje & nie mogą być użyte w kontekście sterty zarządzanej przez GC
- używamy ^ w celu przechowywania uchwytów/referencji do całych obiektów; GC może przesuwać takie obiekty

```
ref class MyClass {  
public:  
    MyClass() {}  
    void Test() { System::Console::WriteLine("hello"); }  
};
```

```
int main() {  
    MyClass ^ p1 = gcnew MyClass;  
    p1->Test();  
    MyClass ^ p2;  
    p2 = p1;  
    p1 = nullptr;  
    p2->Test();  
}
```

Odwołania do zarządzanych typów z niezarządzanego kodu

```
#include <vcclr.h>
using namespace System;

class CppClass {
public:
    gcroot<String^> str;    // can use str as if it were String
    CppClass() {}
};

int main() {
    CppClass c;
    c.str = gcnew String("hello");
    Console::WriteLine( c.str );    // no cast required
}
```

Referencja %

```
ref class MyClass {
public: int i;
};

value struct MyStruct {
int k;
};

int main() {
    MyClass ^ x = gcnew MyClass;
    MyClass ^% y = x;    // adres referencji do obiektu

    int %ti = x->i;    // element obiektu
    int j = 0;
    int %tj = j;    // obiekt na stosie

    int * pi = new int[2];
    int % ti2 = pi[0];    // obiekt na stercie C++
    int *% tpi = pi;    // wskaźnik C++
    MyStruct ^ x2 = gcnew MyStruct;
    MyStruct ^% y2 = x2;
}
```

- przypomina 'standardową' referencję C++,
- referencja % może być deklarowana wyłącznie na stosie,
- 'wskazuje' na obiekty lub składowe,

Słowa kluczowe abstract i sealed

- funkcja abstrakcyjna = pusta funkcja wirtualna
- słowo kluczowe abstract również może być używane przy kompilacji 'standardowej'

```
ref class X abstract {  
public:  
    virtual void f() {};  
    virtual void g() abstract;  
};
```

sealed – w odniesieniu do klasy: nie może być klasą bazową dla żadnej innej; w odniesieniu do funkcji: nie może być przesłaniana w klasie bazowej (przykład 2):

<http://msdn.microsoft.com/en-us/library/0w2w91tf.aspx>

- można dołączyć poprzez `using namespace cli;`
(automatycznie włączona podczas kompilacji `/clr`)
- definiuje słowa kluczowe:
 - `array`
 - `pin_ptr`
 - `interior_ptr`
 - `safe_cast`

Dynamiczne tablice

```
[qualifiers] [cli::]array<[qualifiers]type1[, dimension]>^v  
gcnew [cli::]array<type2[, dimension]>(val[,val...])
```

- `dimension` – liczba wymiarów tablicy, domyślnie 1, maksymalnie 32
- `val` – liczba elementów tablicy (dla każdego wymiaru)
- `type1` – typ elementów tablicy
- `type2` – typ elementów inicjalizujących tablicę (zwykle = `type2`, a w przeciwnym wypadku musi być możliwa konwersja)
- `name` – nazwa zmiennej
- tablice są zarządzanymi obiektami
- brak arytmetyki na indeksach

Jednowymiarowa tablica

```
ref class A {
public:
    int i;
};
array<A^>^ f() {
    int i;
    array<A^>^ ar = gcnew array<A^>(5);

    for (i = 0; i < 5; i++) {
        ar[i] = gcnew A;
        ar[i]->i = i;
    }
    return ar;
}
```

Tablica wielowymiarowa

```
array<Int32, 2>^ Test1() {  
    int i, j;  
    array< Int32,2 >^ local = gcnew array< Int32,2 >(  
        ARRAY_SIZE, ARRAY_SIZE);  
  
    for (i = 0 ; i < ARRAY_SIZE ; i++)  
        for (j = 0 ; j < ARRAY_SIZE ; j++)  
            local[i,j] = i + 10;  
  
    return local;  
}
```

<http://msdn.microsoft.com/en-us/library/2xfh4c7d.aspx>

typedef dla zarządzanej tablicy

```
using namespace System;
ref class G {};

typedef array<array<G^>^> some_array;

int main() {
    some_array ^ MyArr = gcnew some_array (10);
}
```

Tablice tablic (przykład 3):

<http://msdn.microsoft.com/en-us/library/fkzha470.aspx>

Zarządzane tablice jako parametry szablonu

```
using namespace System;
template <class T>
class TA {
public:
    array<array<T>^>^ f() {
        array<array<T>^>^ larr = gcnew array<array<T>^>(10);
        return larr;
    }
};

int main() {
    int retval = 0;
    TA<array<array<Int32>^>^>* ta1 = new TA<array<array<Int32>^>^>
    array<array<array<array<Int32>^>^>^>^ larr = ta1->f();
    retval += larr->Length - 10;
    Console::WriteLine("Return Code: {0}", retval);
}
```

interior ptr

```
cli::interior_ptr<cv_qualifier type> var = &initializer;
```

Przykład:

```
using namespace System;
ref class MyClass { public: int data; };

int main() {
    MyClass ^ h_MyClass = gcnew MyClass;
    h_MyClass->data = 1;
    Console::WriteLine(h_MyClass->data);
    interior_ptr<int> p = &(h_MyClass->data);
    *p = 2;
    Console::WriteLine(h_MyClass->data);
    // alternatywnie:
    interior_ptr<MyClass ^> p2 = &h_MyClass;
    (*p2)->data = 3;
    Console::WriteLine((*p2)->data);
}
```

- używanie podobne do `interior_ptr`, lecz uniemożliwia GC przesuwanie wskazywanego obiektu,
- używamy, gdy chcemy przekazać adres zarządzanego zasobu do niezarządzanego kodu — mamy gwarancje, że adres zarządzanego obiektu znajdującego się na (zarządzanej) stercie nie ulegnie zmianie
- 'związanie' elementu obiektu ma efekt 'związania całego obiektu
- jeśli wartość wskaźnika ulegnie zmianie, to możliwe jest przesuwanie poprzedniego obiektu przez GC
- jeśli sterowanie zostanie przekazane poza blok z deklaracją takiego wskaźnika, to również GC może 'przesuwać' dane

Sortowanie zarządzanej tablicy

- zarządzane tablice dziedziczą 'zachowanie' od klasy bazowej
- przykład to sortowanie tablicy:
 - `Array::Sort(tab);`
 - elementy tablicy muszą implementować metodę `Comparable::CompareTo`

Sortowanie zarządzanej tablicy — przykład

```
value struct Element : public IComparable {
    int v1, v2;

    int CompareTo(Object^ obj) {
        Element^ o = dynamic_cast<Element^>(obj);
        if (o) {
            int thisAverage = (v1 + v2) / 2;
            int thatAverage = (o->v1 + o->v2) / 2;
            if (thisAverage < thatAverage) return -1;
            else if (thisAverage > thatAverage) return 1;
            return 0;
        } else
            throw new ArgumentException("o must be of type 'Element'");
    }
};

int main() {
    array<Element>^ a = gcnew array<Element>(10);
    ...
}
```

delegate

[dostęp] delegate deklaracja_funkcji

```
public delegate void ExDel(int i);

ref class A {
public:    void f(int i) { ... }
          void g(int i) { ... } };

int main () {
    A ^ a = gcnew A;
    ExDel^ d;
    if (d)
        d(7);
    d = gcnew ExDel(a, &A::f);
    d(8);
    DelInst += gcnew ExDel(a, &A::g);
}
```

Szablony a typy generyczne

- szablony są 'styczne', tzn. konkretne instancje typów tworzone są podczas kompilacji kodu,
- instancje dla szablonów tworzone są w trakcie wykonania kodu
- w konsekwencji nie można tworzyć konkretnych instancji szablonów (specjalizacji) mając dostęp do skompilowanego modułu, a takiego ograniczenia nie ma w przypadku typów generycznych

Typy generyczne — przykład

```
using namespace System;
generic <typename ItemType>
ref struct Stack {
    // ItemType may be used as a type here
    void Add(ItemType item) {}
};
generic <typename KeyType, typename ValueType>
ref class HashTable {};

// The keyword class may be used instead of typename:
generic <class ListItem>
ref class List {};

int main() {
    HashTable<int, Decimal>^ g1 = gcnew HashTable<int, Decimal>()
}
```

Opakowania kodu ANSI C++

Sposobem na użycie niezarządzanego kodu C++ z poziomu VB lub C# jest napisanie zarządzanego opakowania dla niezarządzanej klasy. Jeśli klasa niezarządzana nosi nazwę `UClass`, natomiast jej zarządzane opakowanie nazwiemy `MClass`, to wykonujemy następujące kroki:

- 1 w klasie `MClass` deklarujemy zmienną typu `UClass *`;
- 2 dla każdego konstruktora w `UClass` deklarujemy odpowiedni konstruktor w `MClass`, który korzystając z operatora `new` odwoła się do odpowiedniego konstruktora w `UClass`;
- 3 definiujemy destruktor w `MClass`, który usunie obiekt klasy `UClass` korzystając z operatora `delete`;
- 4 dla każdej funkcji składowej w `UClass` definiujemy funkcję w `MClass`, która jedynie wykonuje wywołanie funkcji w `UClass`, dokonując ew. koniecznych konwersji parametrów.

Które funkcje należy 'opakować':

- jeśli funkcja w niezarządzanej klasie jest prywatna, to nie ma potrzeby definiowania odpowiedniej funkcji w klasie zarządzanej;
- nie ma konieczności definiowania opakowań dla funkcji, które nie będą wykorzystywane.

Opakowania kodu ANSI C++, przykład

```
class UA {
    public:
        int f(int i) {
            return i+1;
        }
};

ref class A {
    private:
        UA *ua;
};
```

```
public:
    A() {
        ua = new UA();
    }
    int f( int *i) {
        return ua->f(i);
    }
    ~A() {
        delete ua;
    }
};
```

- PInvoke to zbior usług CLR odpowiedzialnych za:
 - odnalezienie biblioteki z żądaną funkcją
 - załadowanie biblioteki do pamięci
 - odnalezienie adresu f-cji i jej wywołanie wraz z przekazaniem parametrów
- zbiór klas związanych z PInvoke zawarty jest w przestrzeni nazw `System.Runtime.InteropServices`