Dokumentacja Biblioteki insideJS

Robert Bendun

26 marca 2017

Spis treści

1 Wprowadzenie

Biblioteka insideJS umożliwia pisanie natywnego kodu zbliżonego do języka JavaScript w języku C++. Do swojego działania potrzebuje kompilatora obsługującego standard **C++11**. Aktualnie biblioteka **nie jest** prekompilowana. W celu używania biblioteki wystarczy dołączyć plik nagłówkowy "js/index.html". W dalszych przykładach będzie on pomijany. Większość elementów biblioteki umieszczona jest w przestrzeni nazw **js** dlatego w dalszych przykładach będzie ona pomijana. Wyjątkami są makra preprocesora. Wyszczególnione są one w rozdziale "Preprocesor".

2 Tworzenie podstawowych obiektów

Język JavaScript nie posiada jawnie zdefiniowanego terminu klasy. Podstawowym sposobem tworzenia obiektów jest przypisanie odpowiednich pól z ich wartościami. Poniżej przykład kodu:

Listing 1: Podstawowy obiekt

```
#include <iostream>
#include "js/index.hpp"

int main()
{
    using namespace js;

    var object = new Object {
        "name" is "Doge",
        "sound" is "Woooof",
        "age" is 5
    };

    std::cout << object["name"] << '\n'; // returns "Doge"
    std::cout << object["sound"] << '\n'; // returns "Woooof"
    std::cout << object["age"] << '\n'; // returns 5
}
```

Aby dostać się do pól obiektów należy użyć operatora [], gdzie w należy podać odpowiednią nazwę pola. Do wyświetlania pól obiektu można wykorzystywać standardowe strumienie.

Możliwe jest także stworzenie obiektu poprzez funkcję będącą odpowiednikiem konstruktora. Kluczowym elementem takiej funkcji jest zwracanie null na koniec jej działania.

Listing 2: Konstruktor obiektów

```
#include <iostream>
#include "js/index.hpp"

int main()
{
    using namespace js;

    var Dog = new Function(INLINE {
        that["name"] = "Doge";
        that["sound"] = "Woof";
        return null;
    });

    var dog = Dog();

    std::cout << dog["name"] << " says " << dog["sound"];
}
```

Konstruktor może także przyjmować argumenty (należy zwrócić uwagę na zmianę z INLINE na INLINE_ ARG):

Listing 3: Argument konstruktora

```
#include <iostream>
#include "js/index.hpp"

int main()
{
    using namespace js;

    var Dog = new Function(INLINE_ARG {
        that["name"] = *arguments;
        that["sound"] = "Woof";
        return null;
    });

    var dog = Dog("Spike");

    std::cout << dog["name"] << " says " << dog["sound"];
}
```

W przypadku pojedyńczego argumentu należy użyć operatora wyłuskiwania *. W przypadku podania większej liczby argumentów należy użyć operatora indeksu [].

Listing 4: Argumenty konstruktora

```
#include <iostream>
#include "js/index.hpp"

int main()
{
    using namespace js;

    var Pair = new Function(INLINE_ARG {
        that["first"] = arguments[0];
        that["second"] = arguments[1];
        return null;
    });

    var dogs = Pair("Spike", "Doge");
    std::cout << dogs << std::endl;
}
```

3 Funkcje

Aktualnie jedyne funkcje jakie mogą być tworzone są to funkcje lokalne (znajdujące się we wnętrzu klasycznych funkcji z języka C++. Funkcje mogą być tak samo elementem obiektów jak ciągi znaków bądź liczby.

Listing 5: Funkcja wewnątrz obiektu

```
#include <iostream>
#include "js/index.hpp"

int main()
{
    using namespace js;

    var object = new Object {
        "name" is "Doge",
        "sound" is "Woooof",
```

```
        "say" is new Function(INLINE{
            std::cout << that["name"] << " says " << that["sound"] << std::endl;
        })
    };

    object["say"]();
}
```

Słowo kluczowe that jest to kontekst funkcji. Kiedy funkcja jest elementem obiektu jak w powyższym przykładzie that odwołuje się do danego obiektu. Kiedy używamy funkcji konstruktora wtedy that jest odwołaniem do nowego obiektu. Aby uruchomić funkcję z wymuszonym kontekstem należy zastosować metodę call obiektu funkcyjnego.

Listing 6: Ustawianie kontekstu

```
#include <iostream>
#include "js/index.hpp"

int main()
{
    using namespace js;

    var object = new Object {
        "name" is "Doge",
        "sound" is "Woooof"
    };

    var say = new Function(INLINE{
        std::cout << that["name"] << " says " << that["sound"] << std::endl;
    });

    say.call(object);
}
```

Występujące wcześniej INLINE oraz INLINE_ ARG są w rzeczywistości następującymi deklaracjami preprocesora.

Listing 7: Deklaracje preprocesora

```
#define INLINE [&](js::iterator&& that)->js::Atom*
#define INLINE_ARG [&](js::iterator&& that, js::iterator &&arguments)->js::Atom*
```

Klasa iterator jest to wraper przechowujący wskaźnik do rzeczywistego obiektu/funkcji. Umożliwa takie operacje jak wyłuskanie, wywołanie czy indeksowanie. Więcej informacji znajduje się w rozdziale "Przechowywanie obiektów".

4 Tablice

Tablice tworzone są w sposób analogiczny do obiektów. Wyjątkiem jest sposób nadawania identyfikatorów. Występuje on niejawnie.

Listing 8: Tworzenie tablicy

```
#include <iostream>
#include "js/index.hpp"

int main()
{
    using namespace js;

    var table = new Array{ "Hello", 10, "World", 20, 30 };
    std::cout << table[0] << std::endl; // returns "Hello"
    std::cout << table << std::endl; // returns ["Hello", 10, "World", 20, 30]
}
```

Jest wiele sposobów na dostanie się do elementów kontenera. Aby iterować po elementach tablicy można wykorzystać:

- pętle zakresową for z C++11
- tradycyjną pętlę for
- metodę for_ each (a dokładnie to makro ułatwiające korzystanie z niej

Wszystkie z poniższych metod pokazano poniżej. Rezultatem każdej z nich jest ciąg znaków "Hello 10 World 20".

Listing 9: Iteracja po tablicy

```
#include "js/index.hpp"

int main()
{
    using namespace js;

    var table = new Array{ "Hello", 10, "World", 20 };
    std::string result1, result2, result3;

    // 1
    for(auto &x : cast<Array>(table))
        result1 += x->toString() + ' ';

    // 2
    cast<Array>(table).forEach({
        result2 += element->toString() + ' ';
    });

    // 3
    for(std::size_t i=0; i < cast<Array>(table).length(); i++)
        result3 += table[i]->toString() + ' ';

}
```

Powyższy przykład można wykonać dużo prościej za pomocą metody join.

Listing 10: Wykorzystywanie metody join

```
#include "js/index.hpp"

int main()
{
    using namespace js;
    var table = new Array{ "Hello", 10, "World", 20 };
    std::string result = cast<Array>(table).join(" ");
}
```

Na tablicy można wykonać również następujące operacje:

- sortowanie za pomocą metody sort()
- odwracanie tablicy za pomocą metody reverse()
- dodawanie elementów na koniec za pomocą push()
- łączenie tablic za pomocą metody concat()

Listing 11: Pozostałe metody operujące na tablicach

```
#include "js/index.html"

int main()
{
    using namespace js;

    var table = new Array{ 39, 10, 12, 20 };
    var table_reversed = cast<Array>(table).reverse();

    auto compare = [](iterator leftIt, iterator rightIt)->bool {
        if((*leftIt)->type() == "number" && (*rightIt)->type() == "number") {
            auto left = cast<Number>(leftIt);
            auto right = cast<Number>(rightIt);
            return ( left->get() < right->get() );
        } else return false;
    };

    var table_sorted = cast<Array>(table).sort(compare);

    var small_1 = new Array{3, 4};
    var small_2 = new Array{"Hello", "World"};

    var table_concat = cast<Array>(table).concat(small_1, small_2);
}
```

Listings