

# Podstawy programownia (w języku C++)

## Pierwsze kroki (płytką wodą)

Marek Marecki

Polsko-Japońska Akademia Technik Komputerowych

18 października 2020

## Wskaźnik i tablica w stylu C

# TRADYCYJNY PIERWSZY PROGRAM

Tradycją w świecie programistów jest, aby pierwszym programem napisanym w nowym języku było wypisanie na ekran tekstu "Hello, World!". W C++ taki program wygląda następująco:

```
#include <iostream>

auto main() -> int
{
    std::cout << "Hello, World!\n";
    return 0;
}
```

Na jego przykładzie omówię strukturę programu.

# PLIKI NAGŁÓWKOWE

```
#include <iostream>

auto main() -> int
{
    std::cout << "Hello, World!\n";
    return 0;
}
```

Pliki nagłówkowe zawierają deklaracje funkcji, zmiennych, stałych, oraz struktur danych. Bez "podłączenia" nagłówków do naszego kodu źródłowego kompilator nie będzie świadomy istnienia rzeczy zdefiniowanych w tych plikach i nie będzie możliwe ich użycie.

Sposobem na dołączenie plików nagłówkowych do naszego kodu źródłowego jest *dyrektywa* `#include`.

# NAZWA FUNKCJI

## DEFINICJA FUNKCJI

```
#include <iostream>

auto main() -> int
{
    std::cout << "Hello, World!\n";
    return 0;
}
```

Definicję funkcji rozpoczyna *słowo kluczowe* `auto`, po którym podana jest nazwa funkcji – np. `main`.

Nazwa funkcji może składać się wyłącznie z małych i wielkich liter, znaków podkreślenia (`_`) i cyfr, oraz **musi** rozpocząć się literą.

W szkole na matematyce pisało się  $f(x) = x + 1$ . Nazwą funkcji było oczywiście  $f$ .

# LISTA PARAMETRÓW FORMALNYCH FUNKCJI

## DEFINICJA FUNKCJI

```
#include <iostream>

auto main() -> int
{
    std::cout << "Hello, World!\n";
    return 0;
}
```

Po nazwie podana jest lista *parametrów formalnych* funkcji, czyli wartości, których podania funkcja będzie wymagała przy jej wywołaniu. Lista ta może być pusta.

W szkole na matematyce pisało się  $f(x) = x + 1$ . Parametrem formalnym funkcji  $f$  jest w tym przypadku  $x$ .

# TYP ZWRACANY

## DEFINICJA FUNKCJI

```
#include <iostream>

auto main() -> int
{
    std::cout << "Hello, World!\n";
    return 0;
}
```

Kolejnym elementem jest *typ zwracany* (ang. *return type*) funkcji, czyli typ wartości produkowanych przez daną funkcję. Zapisuje się go po "strzałce". Jeśli funkcja nie produkuje żadnych wartości należy użyć typu `void`.

W szkole na matematyce pisało się  $f(x) = x + 1$  i nie podawało się typu zwracanego. W domyśle typem tym była "liczba".

# DEKLARACJA FUNKCJI

## DEFINICJA FUNKCJI

```
#include <iostream>

auto main() -> int;
{
    std::cout << "Hello, World!\n";
    return 0;
}
```

Gdyby po typie zwracanym wpisać średnik powstałaby *deklaracja funkcji*. Takie deklaracje umieszcza się w plikach nagłówkowych. Są one deklaracją tego, że definicja takiej funkcji będzie dla kompilatora dostępna i powinien on pozwolić na jej użycie.

W szkole na matematyce pisało się  $f(x) = x + 1$  i nie występowało coś takiego jak deklaracja.



# CIAŁO FUNKCJI 1

## DEFINICJA FUNKCJI

```
#include <iostream>

auto main() -> int
{
    std::cout << "Hello, World!\n";
    return 0;
}
```

Po typie zwracanym zapisuje się *ciało funkcji*, czyli grupę instrukcji, których dana funkcja jest abstrakcją<sup>1</sup>. Ciało funkcji musi być otoczone nawiasami klamrowymi.

W szkole na matematyce pisało się  $f(x) = x + 1$  i trzeba było otaczać ciała funkcji (czyli  $x + 1$ ) nawiasami klamrowymi.

---

<sup>1</sup>patrz *procedural abstraction* z poprzedniego wykładu

# CIAŁO FUNKCJI 2

## DEFINICJA FUNKCJI

```
#include <iostream>

auto main() -> int
{
    std::cout << "Hello, World!\n";
    return 0;
}
```

Ciało funkcji może składać się z dowolnej liczby instrukcji.

W szkole na matematyce pisało się  $f(x) = x + 1$ , a ciałem funkcji było zazwyczaj jakieś proste działanie.

# CIAŁO FUNKCJI 3

## DEFINICJA FUNKCJI

```
#include <iostream>

auto main() -> int
{
    std::cout << "Hello, World!\n";
    return 0;
}
```

Jeśli funkcja produkuje jakąś wartość (jej typem zwracanym nie jest void), to **musi** pojawić się w jej ciele instrukcja return.

Typ wartości zwróconej przez instrukcję return **musi** się zgadzać z typem zwracanym deklarowanym przez funkcję.

W szkole na matematyce pisało się  $f(x) = x + 1$  i oczywistym było, że zwracaną wartością jest wynik dodawania.

# NAZWA FUNKCJI

"W SZKOLE NA MATEMATYCE" vs C++

f

VS

auto f

# LISTA PARAMETRÓW FORMALNYCH

"W SZKOLE NA MATEMATYCE" vs C++

`f(x)`

VS

`auto f(int const x)`

# TYP ZWRACANY

"W SZKOLE NA MATEMATYCE" vs C++

$f(x)$

VS

```
auto f(int const x) -> int
```

# CIAŁO FUNKCJI

"W SZKOLE NA MATEMATYCE" vs C++

$f(x) = x + 1$

VS

```
auto f(int const x) -> int
{
    return (x + 1);
}
```

# NAZWA FUNKCJI

"W SZKOLE NA MATEMATYCE" vs C++

```
main
```

VS

```
auto main
```



# LISTA PARAMETRÓW FORMALNYCH

"W SZKOLE NA MATEMATYCE" vs C++

```
main()
```

VS

```
auto main()
```

# TYP ZWRACANY

"W SZKOLE NA MATEMATYCE" vs C++

```
main()
```

VS

```
auto main() -> int
```

# CIAŁO FUNKCJI

"W SZKOLE NA MATEMATYCE" vs C++

```
main(x) = print("Hello, World!"), 0
```

VS

```
auto main() -> int  
{  
    std::cout << "Hello, World!\n";  
    return 0;  
}
```

## OPERATOR PRZEKIEROWANIA: <<

Wracając do przykładu ze slajdu 3. Może nie być jasne co dzieje się w tej linijce:

```
std::cout << "Hello , World!\n";
```

Otóż...

## OPERATOR PRZEKIEROWANIA: <<

Do zmiennej globalnej `std::cout`

```
std::cout << "Hello, World!\n";
```

...za pomocą *operatora przekierowania*

```
std::cout << "Hello, World!\n";
```

...wysyłany jest napis `Hello World!\n` (w C++ napisy ograniczane są znakami cudzysłowu).

```
std::cout << "Hello, World!\n";
```

Co spowoduje wypisanie tego napisu na ekran. Można w ten sposób wypisać na ekran różne wartości (liczbowe, logiczne, itd.).

## PRZYKŁADOWY KOD

Kod dla programu Hello, World! znajduje się w repozytorium z zajęciami<sup>2</sup> w pliku 'src/00-hello-world.cpp' Można go skompilować następującym poleceniem:

```
make build/00-hello-world.bin
```

Uruchomienie:

```
./build/00-hello-world.bin
```

Zadanie: zmienić kod tak, żeby wypisywał Hello, a potem imię studenta.

---

<sup>2</sup><https://git.sr.ht/~maelkum/education-introduction-to-programming-cxx>

# OVERVIEW

Pierwszy program

Argumenty wiersza poleceń

Wskaźnik i tablica w stylu C

## ARGUMENTY DO PROGRAMU

W repozytorium z zajęciami znajduje się plik 'src/01-hello-argv.cpp'. Zawiera on kod źródłowy programu, który używa argumentów przekazanych mu na wierszu poleceń.

Można go skompilować następującym poleceniem:

```
make build/00-hello-argv.bin
```

Uruchomienie:

```
./build/01-hello-argv.bin Kasia
```

Zadanie: sprawdzić się stanie jak nie poda się argumentu (tj., uruchomi program bez "Kasia").



# PARAMETRY FORMALNE FUNKCJI MAIN

## ARGUMENTY WIERSZA POLECEŃ

Żeby mieć możliwość odczytania argumentów podanych do programu na wierszu poleceń, lista parametrów formalnych funkcji main musi wyglądać następująco:

```
auto main(int argc, char* argv[]) -> int
```

Na kolejnych slajdach objaśnię znaczenie każdego z tych parametrów.

# PARAMETRY FORMALNE FUNKCJI MAIN – argc

## ARGUMENTY WIERSZA POLECEŃ

```
auto main(int argc, char* argv[]) -> int
```

argc (od *argument count*) przechowuje liczbę argumentów przekazanych do programu jako liczbę całkowitą.

# PARAMETRY FORMALNE FUNKCJI MAIN – argv

## ARGUMENTY WIERSZA POLECEŃ

```
auto main(int argc, char* argv[]) -> int
```

argv (od *argument values*) przechowuje wartości argumentów przekazanych do programu. Typ parametru argv może być nieco zagdkowy, ale ta zagadka zostanie rozwiązana na kolejnych slajdach.

# PARAMETR `argv` – TABLICA WSKAŹNIKÓW DO `char`

ARGUMENTY WIERSZA POLECEŃ

Parametr `argv` jest

```
char* argv[]
```

...tablicą w stylu C (unikamy ich)

```
char* argv[]
```

...wskaźników (oznaczanych przez `*` za nazwą typu) do `char`

```
char* argv[]
```

`char*` jest sposobem na reprezentację napisów w stylu C.

# OVERVIEW

Pierwszy program

Argumenty wiersza poleceń

Wskaźnik i tablica w stylu C

Wskaźnik

Tablica w stylu C

# THE GOOD, THE BAD, AND THE UGLY ...W STYLU C

Język C++ wywodzi się z języka C. Jeśli dla jakiejś konstrukcji język C++ oferuje swój zamiennik, to ta odziedziczona jest określana jako "w stylu C" (ang. *C-style*).

# THE UGLY - WSKAŹNIKI

THE GOOD, THE BAD, AND THE UGLY ...W STYLU C

Konstrukcją "brzydką" są wskaźniki.

Ich typy są nieintuicyjne w zapisie, a wskaźniki same w sobie nie oferują żadnej gwarancji poprawności - nigdy nie wiemy czy wskaźnik przypadkiem nie jest *wiszący*<sup>3</sup>.

W C++ część zadań wskaźników przejęły *referencje* (ang. *reference*).

---

<sup>3</sup>wytłumaczenie na slajdzie 37

# THE BAD - TABLICE W STYLU C

## THE GOOD, THE BAD, AND THE UGLY ...W STYLU C

Konstrukcją "złą" są tablice.

Bardzo łatwo gubią rozmiar (który dla pewności musi być przechowywany w osobnej zmiennej), a jeśli zostanie on zgubiony niemożliwe jest jego odtworzenie. Na dodatek, tablice bardzo "chętnie" degradują się do wskaźników tracąc całkowicie informację o tym, że przechowują  $n$  elementów i nabywając wszystkie wady wskaźnika.

W C++ zamiennikiem (dużo lepszym) tablic w stylu C są: `std::array` (dla tablic o stałym rozmiarze) i `std::vector` (dla tablic o zmiennym rozmiarze).



# WSKAŹNIK

Czym jest wskaźnik<sup>4</sup>? Wskaźnik jest adresem fragmentu pamięci (zawierającego dane typu  $t$ ).

Mając wskaźnik można "dostać się" do danych umieszczonych w pamięci pod adresem, na który wskazuje wskaźnik.

Typ wskaźnika zapisuje się jako:  $t^*$

gdzie  $t$  jest typem danych leżących pod adresem, na który wskazuje wskaźnik.

Typem oznaczającym "wskaźnik do `int`" będzie `int*`

---

<sup>4</sup>Część "Data structures" na poprzednim wykładzie.

# Tworzenie wskaźników

## Wskaźnik

Aby otrzymać adres fragmentu pamięci zawierającego zmienną `x` używa się operatora `&` (ampersand)

```
auto x          = int{42};  
auto x_pointer = &x;
```

Wskaźnik otrzymujemy również w sytuacji gdy alokujemy pamięć dynamicznie.

# UŻYWANIE WSKAŹNIKÓW

## WSKAŹNIK

Aby użyć (odczytać lub zmodyfikować) danych z fragmentu pamięci wskazywanego przez wskaźnik używa się operatora \* (gwiazdka)

```
auto x          = int{42};    // x contains 42
auto x_pointer = &x;    // x_pointer contains address of x

auto y          = (*x_pointer + 1);    // y contains 43
*x_pointer = 44;    // x contains 44
```

# PO CO SĄ WSKAŹNIKI?

## WSKAŹNIK

Wskaźniki są niezbędne jeśli chcemy korzystać z dynamicznej alokacji pamięci. Jeśli jakaś wartość jest alokowana w czasie wykonywania programu, jest ona umieszczana w innym obszarze pamięci (na *stercie*, ang. *heap*) niż wartości alokowane podczas kompilacji (na *stosie*, ang. *stack*) i niemożliwy byłby bezpośredni dostęp do niej.

Wskaźniki są niezbędne również jeśli chcemy przekazać dane jako argument do funkcji, ale ich kopiowanie (domyślny sposób przekazywania argumentów) byłoby kosztowne. W takim wypadku przekazujemy funkcji jedynie adres danych i pozwalamy jej używać ich "w miejscu".

# PROBLEMY ZE WSKAŹNIKAMI

## WSKAŹNIK

Wskaźniki mogą być *zerowe* (ang. *null pointer*, słowo kluczowe `nullptr`), czyli wskazywać na adres 0 w pamięci.

Wskaźniki mogą być *wiszące* (ang. *dangling pointer*), czyli wskazywać na adres w pamięci, który już nie należy do naszego programu (np. został zdealokowany i oddany do systemu operacyjnego).

Jeśli spróbujemy użyć wiszącego lub zerowego wskaźnika do odczytania lub modyfikacji danych nasz komputer może wybuchnąć i mogą nam "z nosa wylecieć demony"<sup>5</sup>. Jest to tzw. *zachowanie niezdefiniowane* (ang. *undefined behaviour*) - jest ono źródłem wielu awarii i naruszeń bezpieczeństwa w programach.

---

<sup>5</sup>[https://en.wikipedia.org/wiki/Undefined\\_behavior](https://en.wikipedia.org/wiki/Undefined_behavior)

## TABLICA W STYLU C

Czym jest tablica w stylu C<sup>6</sup>? Tablica w stylu C jest fragmentem pamięci wystarczająco dużym żeby pomieścić  $n$  wartości typu  $t$ .

Typ tablicy zapisuje się jako:  $t[n]$  ( $n$  można pominąć)  
gdzie  $t$  jest typem danych leżących pod adresem, na który wskazuje tablica, a  $n$  rozmiarem tablicy.

Typem oznaczającym "tablicę *nie-wiadomo-ilu* wartości `int`" będzie `int []`

Typem oznaczającym "tablicę 4 wartości `int`" będzie `int [4]`

---

<sup>6</sup>Część "Data structures" na poprzednim wykładzie.

# Tworzenie tablic w stylu C

## Tablica w stylu C

Aby stworzyć tablicę  $n$  wartości typu  $t$  używa się następującej składni:

```
t array[n];
```

Dla przykładu, tablica 4 wartości typu `int`:

```
int array[4];
```

Kompilator może też określić rozmiar tablicy automatycznie jeśli podamy elementy jakimi tablica powinna zostać zainicjalizowana:

```
int array[] = { 0, 1, 2, 3 };
```

# UŻYWANIE TABLIC W STYLU C

## TABLICA W STYLU C

Aby dostać się do elementu  $n$  w tablicy używa się operatora []

```
int array[] = { 42, 64, 127, -1 };  
auto x = array[0]; // x contains 42
```



# PO CO SĄ TABLICE?

## TABLICA W STYLU C

Tablice służą do przechowywania wielu wartości tego samego typu.

W C++ dostępne są lepsze zamienniki tablic: `std::array` i `std::vector`.