

# Podstawy programowania (w języku C++)

## Pętle

Marek Marecki

Polsko-Japońska Akademia Technik Komputerowych

4 października 2021

# OVERVIEW

while i do-while

for

Zadania (pętle)

range-based for

Podsumowanie

# Po co?

## PĘTLA while

Pętla while sprawdza się kiedy instrukcja przez nią wykonywana powinna być powtarzana *dopóki* pewien warunek jest spełniony.

```
while (system_is_running()) {  
    process_events();  
}
```

Istotne jest to, że warunek sprawdzany jest *przed* wykonaniem instrukcji.

# Po co?

## PĘTLA do-while

Pętla do-while sprawdza się kiedy instrukcja przez nią wykonywana powinna być powtarzana *dopóki* pewien warunek jest spełniony, ale musi być wykonana *co najmniej jeden raz*.

```
do {  
    process_events();  
} while (system_is_running());
```

Istotne jest to, że warunek sprawdzany jest *po* wykonaniu instrukcji.

# WARUNEK

PĘTLE while i do-while

Warunek jest podawany w nawiasach po słowie kluczowym while, i może być w zasadzie dowolny.

```
while (system_is_running()) {  
    process_events();  
}
```

albo

```
do {  
    process_events();  
} while (system_is_running());
```

# INSTRUKCJA

## PĘTLE while i do-while

Instrukcja powtarzana przez pętlę jest podawana w nawiasach klamrowych:

```
while (system_is_running()) {  
    process_events();  
}
```

albo

```
do {  
    process_events();  
} while (system_is_running());
```

# AD INFINITUM

## PĘTLE while i do-while

Do implementacji pętli nieskończonych często wykorzystuje się konstrukcję `while-true`:

```
while (true) {  
    process_events();  
}
```

Pętle nieskończone są często spotykane w "sercach" długo działających programów (systemów operacyjnych, gier, itp.), których zakończenie jest wywoływane przez jakieś zewnętrzne zdarzenie (np. akcję użytkownika), a nie przez wewnętrzny stan programu (np. koniec danych do przetworzenia).

# KROK PO KROKU

PĘTLE while i do-while

while

VS

while



# KROK PO KROKU

PĘTLE while i do-while

```
while (condition_is_met())
```

VS

```
while (condition_is_met());
```

# KROK PO KROKU

PĘTLE while i do-while

```
while (condition_is_met()) {  
    take_action();    // maybe never  
}
```

VS

```
do {  
    take_action();    // at least once  
} while (condition_is_met());
```

# OVERVIEW

while i do-while

for

Zadania (pętle)

range-based for

Podsumowanie

# Po co?

## PĘTLA for

Pętla for sprawdza się kiedy instrukcja przez nią wykonywana powinna być powtarzana pewną *ilość razy* określoną przez licznik pętli.

```
std::cout << argv[0];  
for (auto i = 1; i < argc; ++i) {  
    std::cout << " " << argv[i];  
}  
std::cout << "\n";
```

Istotne jest to, że warunek sprawdzany jest *przed* wykonaniem instrukcji.

# INICJALIZACJA LICZNIKA

## PĘTLA for

Licznik jest inicjalizowany *wewnątrz* pętli, wewnątrz nawiasów po słowie kluczowym for:

```
std::cout << argv[0];  
for (auto i = 1; i < argc; ++i) {  
    std::cout << " " << argv[i];  
}  
std::cout << "\n";
```

# WARUNEK

## PĘTLA for

Warunek zapisywany jest po średniku kończącym inicjalizację licznika:

```
std::cout << argv[0];  
for (auto i = 1; i < argc; ++i) {  
    std::cout << " " << argv[i];  
}  
std::cout << "\n";
```

Warunek, tak jak w pętli while, jest sprawdzany przed wykonaniem instrukcji powtarzanej przez pętlę.

# KROK

## PĘTLA for

Krok jest wykonywany *po* instrukcji powtarzanej w pętli, i służy do aktualizacji licznika:

```
std::cout << argv[0];  
for (auto i = 1; i < argc; ++i) {  
    std::cout << " " << argv[i];  
}  
std::cout << "\n";
```

# INSTRUKCJA

## PĘTLA for

Instrukcja powtarzana przez pętlę jest zapisywana w nawiasach klamrowych:

```
std::cout << argv[0];  
for (auto i = 1; i < argc; ++i) {  
    std::cout << " " << argv[i];  
}  
std::cout << "\n";
```



# KROK PO KROKU

PĘTLA for

for

# KROK PO KROKU

PĘTLA for

```
for (auto i = 1;;)
```

# KROK PO KROKU

PĘTLA for

```
for (auto i = 1; i < argc;)
```

# KROK PO KROKU

## PĘTLA for

```
for (auto i = 1; i < argc; ++i)
```

# KROK PO KROKU

## PĘTLA for

```
for (auto i = 1; i < argc; ++i) {  
    std::cout << " " << argv[i];  
}
```

# OVERVIEW

while i do-while

for

Zadania (pętle)

range-based for

Podsumowanie

## ZADANIE: HASŁO

Program, który jako argument na wierszu poleceń pobierze napis (hasło), a potem będzie użytkownika prosił w pętli o ponowne podanie tego hasła dopóki nie zostanie ono wpisane poprawnie. Dla przykładu<sup>1</sup>:

```
./build/s03-password.bin student
password: profesor
password: dziekan
password: student
ok!
```

Kod źródłowy w pliku `src/s03-password.cpp`

---

<sup>1</sup> na zielono rzeczy wpisywane przez użytkownika

## ZADANIE: ODLICZANIE

Program, który jako argument na wierszu poleceń pobierze liczbę i rozpocznie odliczanie od niej (włącznie) do zera (włącznie). Dla przykładu:

```
./build/s03-countdown.bin 3  
3...  
2...  
1...  
0...
```

Kod źródłowy w pliku `src/s03-countdown.cpp`



## ZADANIE: GRA W ZGADYWANIE

Program, który wylosuje<sup>2</sup> liczbę całkowitą od 1 do 100 i będzie prosić użytkownika o zgadnięcie tej liczby. Po nieudanej próbie program powinien wyświetlić wskazówkę (np. "za mała liczba", "za duża liczba").

```
./build/s03-guessing-game.bin  
guess: 10  
number too small!  
guess: 90  
number too big!  
guess: 50  
just right!
```

Kod źródłowy w pliku `src/s03-guessing-game.cpp`

---

<sup>2</sup>patrz slajd 44. z pierwszego wykładu

## ZADANIE: FIZZBUZZ

Program, który wczyta podaną jako argument na wierszu poleceń liczbę, a następnie dla każdego  $n$  w zakresie od 1 (włącznie) do tej liczby (włącznie) wykona następujące rzeczy:

1. wypisze  $n$  na ekran
2. jeśli  $n$  jest podzielne przez 3 wypisze "Fizz" (np. 3 Fizz)
3. jeśli  $n$  jest podzielne przez 5 wypisze "Buzz" (np. 5 Buzz)
4. jeśli  $n$  jest podzielne przez 3 i 5 wypisze "FizzBuzz" (np. 15 FizzBuzz)

To czy liczba  $a$  jest podzielna przez  $n$  można sprawdzić operatorem  $\%$  (*modulo*) zwracającym resztę z dzielenia; ' $a \% n$ ' zwróci resztę z dzielenia  $a$  przez  $n$ .

Kod źródłowy w pliku `src/s03-fizzbuzz.cpp`

## ZADANIE: echo(1)

Program, który wypisze argumenty podane mu na wierszu poleceń. Wypisane argumenty muszą być oddzielone znakiem spacji.

Kod źródłowy w pliku `src/s03-echo.cpp`

Ćwiczenie dodatkowe:

1. jeśli na początku pojawi się opcja `-n` nie drukować znaku nowej linii na końcu programu
2. jeśli na początku pojawi się opcja `-r` wydrukować argumenty w odwrotnej kolejności
3. jeśli na początku pojawi się opcja `-l` wydrukować argumenty po jednym na linię
4. obsłużyć sytuację, w której jednocześnie podane są opcje `-r -l` albo `-r -n`

## ZADANIE: 99 BOTTLES OF BEER

Program powinien w pętli wyświetlić tekst piosenki<sup>3</sup>. Rozpoczynając od 99 (lub liczby podanej jako argument na wierszu poleceń) program ma wypisać:

*99 bottles of beer on the wall, 99 bottles of beer.*

*Take one down, pass it around, 98 bottles of beer on the wall...*

Po osiągnięciu 0 program ma wypisać:

*No more bottles of beer on the wall, no more bottles of beer.*

*Go to the store and buy some more, 99 bottles of beer on the wall...*

i zakończyć pracę.

Kod źródłowy w pliku `src/s03-beer.cpp`

---

<sup>3</sup>[https://en.wikipedia.org/wiki/99\\_Bottles\\_of\\_Beer](https://en.wikipedia.org/wiki/99_Bottles_of_Beer)

# OVERVIEW

while i do-while

for

Zadania (pętle)

range-based for

Podsumowanie

# Po co?

## PĘTLA RANGE-BASED for

Pętla *range-based* for sprawdza się kiedy instrukcja przez nią wykonywana powinna być powtórzona dla *każdego elementu* pewnej wartości.

```
for (auto const& each : employees) {  
    pay_salary(each);  
}
```

Kompilator języka C++ automatycznie wygeneruje kod, który będzie odpowiedzialny za sprawdzenie warunku końca iteracji.

# ELEMENT

## PĘTLA RANGE-BASED for

Zmienna (lub stała), która reprezentuje aktualny element definiowana jest *wewnątrz* pętli, wewnątrz nawiasów po słowie kluczowym for:

```
for (auto const& each : employees) {  
    pay_salary(each);  
}
```

Jeśli pętla ma za zadanie zmodyfikować elementy trzeba użyć zapisu  $T\&$  czyli *referencja do  $T$* <sup>4</sup>.  
Jeśli modyfikacja elementów jest niepożądana, warto użyć zapisu  $T \text{ const}$ , czyli *stała typu  $T$* .  
Kompilator nie pozwoli na modyfikację takich wartości.

Jeśli tworzenie kopii elementów jest kosztowne, a ich modyfikacje niepożądane można połączyć te dwa zapisy w  $T \text{ const}\&$ , czyli *referencja do stałej typu  $T$* .

---

<sup>4</sup>referencja to taki wskaźnik, który udaje, że nie jest wskaźnikiem i poprawia komfort życia programisty

# ZAKRES

## PĘTLA RANGE-BASED for

Zakres iteracji jest określony przez pewną wartość, podaną po dwukropku:

```
for (auto const& each : employees) {  
    pay_salary(each);  
}
```

Wartość ta może być zmienną, stałą, a nawet być wynikiem wywołania funkcji.



# INSTRUKCJA

## PĘTLA RANGE-BASED for

Instrukcja podawana jest w nawiasach klamrowych i wykorzystuje element:

```
for (auto const& each : employees) {  
    pay_salary(each);  
}
```

# KROK PO KROKU

## PĘTLA RANGE-BASED for

for

# KROK PO KROKU

## PĘTLA RANGE-BASED for

```
for (auto const& each_element : )
```

# KROK PO KROKU

## PĘTLA RANGE-BASED for

```
for (auto const& each_element : some_value)
```

# KROK PO KROKU

## PĘTLA RANGE-BASED for

```
for (auto const& each_element : some_value) {  
    use_element_to_do_stuff(each_element);  
}
```

# NAJPROSTSZY PRZYKŁAD Z MOŻLIWYCH

PĘTLA RANGE-BASED for

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>

auto main(int argc, char* argv[]) -> int
{
    auto args = std::vector<std::string>{};
    std::copy_n(argv, argc, std::back_inserter(args));

    for (auto const& each : args) {
        std::cout << each << "\n";
    }

    return 0;
}
```

# NAJPROSTSZY PRZYKŁAD Z MOŻLIWYCH

## PĘTLA RANGE-BASED for

Ten fragment tworzy *wektor*<sup>5</sup> z tablicy argumentów, przekazanych funkcji `main()` na wierszu poleceń.

```
auto args = std::vector<std::string>{};
std::copy_n(argv, argc, std::back_inserter(args));
```

Funkcja `std::copy_n` (kopiująca `argc` elementów z tablicy `argv`) pochodzi z nagłówka `algorithm`, a funkcja `std::back_inserter` (dodająca elementy do `args`) z nagłówka `iterator`.

---

<sup>5</sup><https://en.cppreference.com/w/cpp/container/vector>

# NAJPROSTSZY PRZYKŁAD Z MOŻLIWYCH

## PĘTLA RANGE-BASED for

Następnie argumenty zebrane w zmiennej `args` wypisywane są po kolei na standardowy strumień wyjścia.

```
for (auto const& each : args) {  
    std::cout << each << "\n";  
}
```



# KALKULATOR

## PĘTLA RANGE-BASED for

Używając pętli *range-based* for oraz biblioteki standardowej można szybko napisać własny kalkulator obliczający wyrażenia zapisane w *odwrotnej notacji polskiej*<sup>6</sup>:

```
make build/04-rpn-calculator.bin
./build/04-rpn-calculator.bin 2 2 + p
4
```

Kod źródłowy kalkulatora znajduje się w repozytorium z szablonem zajęć:

<https://git.sr.ht/~maelkum/education-introduction-to-programming-cxx/tree/master/src/04-rpn-calculator.cpp>

---

<sup>6</sup>[https://en.wikipedia.org/wiki/Reverse\\_Polish\\_notation](https://en.wikipedia.org/wiki/Reverse_Polish_notation)

# KALKULATOR – ODWROTNA NOTACJA POLSKA

PĘTLA RANGE-BASED for

Odwrotna notacja polska jest notacją *postfiksową* (ang. *postfix*), czyli *operator* występuje po *operandach*:  $2\ 2\ +$

Typowa, znana ze szkolnej matematyki, notacja z operatorem pomiędzy operandami to notacja *infiksowa* (ang. *infix*):  $2\ +\ 2$

Wywołania funkcji są zapisywane w notacji polskiej, prefiksowej (ang. *prefix*) - czyli z operatorem zapisywanym przed operandami:  $+ 2\ 2$

# ZADANIE

## PĘTLA RANGE-BASED for

Rozwinąć kalkulator z poprzednich slajdów o funkcje:

1. mnożenia, operatorem \*
2. dzielenia, operatorem /
3. dzielenia liczb całkowitych, operatorem // (czyli '5 2 //' da 2, a nie 2.5)
4. reszty z dzielenia, operatorem %
5. potęgowania, operatorem \*\*<sup>7</sup>
6. pierwiastka kwadratowego, operatorem sqrt
7. jednej operacji wymyślonej przez siebie

Kod źródłowy w pliku `src/s04-rpn-calculator.cpp`

---

<sup>7</sup>operatory zawierające znak \* trzeba na wierszu poleceń "otoczyć" znakiem apostrofu żeby powłoka nie potraktowała ich jako znaków specjalnych, np. `2 2 '*'`

# OVERVIEW

while i do-while

for

Zadania (pętle)

range-based for

Podsumowanie

# PODSUMOWANIE

Student powinien umieć:

1. wykorzystać pętle `while`, `do-while`, `for`, oraz *range-based for*
2. wykorzystać liczby losowe w programie

# ZADANIA

## PODSUMOWANIE

Zadania znajdują się na slajdach 23, 24, 25, 26, 27, 28, 43.