Podstawy programownia (w języku C++)

Pętle i struktury danych

Marek Marecki

Polsko-Japońska Akademia Technik Komputerowych

28 października 2020

Overview

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* wykonaniem 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

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

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[1]:

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

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

[1] 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[2] 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`.

[2] 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 odzielone 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`

Overview

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[3].
Jeśli modyfikowacja elementów jest niepożądana, warto użyć zapisu T `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 `const&`, czyli *referencja do stałej typu* T.

[3] 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*[4] 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`.

[4] `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*[5]:

```
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`

[5] `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 `**`[6]
6. pierwiastka kwadratowego, operatorem `sqrt`
7. jednej operacji wymyślonej przez siebie

[6] operatory zawierające znak * trzeba na wierszu poleceń "otoczyć" znakiem apostrofu żeby powłoka nie potraktowała ich jako znaków specjalnych, np. `2 2 '*'`

Overview

Podsumowanie

Student powinien umieć:

1. wykorzystać pętle `while`, `do-while`, `for`, oraz *range-based* `for`
2. samodzielnie zaprojektować własny typ danych, jego pola i funkcje składowe
3. wytłumaczyć czym jest i jak działa funkcja składowa, oraz czym jest `this`
4. powiedzieć jaka jest rola konstruktora i destruktura
5. wykorzystać liczby losowe w programie

Zadania

Podsumowanie

Zadania znajdują się na slajdach 23, 24, 25, 26, 27, 42.