Podstawy programownia (w języku C++)

Wstęp do C++

Marek Marecki

Polsko-Japońska Akademia Technik Komputerowych

4 października 2021

Overview

Język C++

Piękno, harmonia, wdzięk, dobry rytm - wszystkie zależą od prostoty.

Platon, *Republika*

If you think it's simple, then you have misunderstood the problem.

Bjarne Stroustrup, autor języka C++

C++ jest rozbudowanym językiem oferującym wiele możliwości. Ceną za to jest jego ogromne skomplikowanie, brak "elegancji", i pułapki czychające na nieuważnego programistę.

Overview

C++

Control flow

Data types

I/O

SEQUENCE

Control flow

Kontrola w języku C++ zaczyna się od pierwszej instrukcji w funkcji `main()`. Kolejne instrukcje wykonywane są w kolejności zdefiniowanej w kodzie źródłowym, a większość z nich zakończona jest operatorem `;`

Grupy instrukcji są ograniczane nawiasami klamrowymi `{ }` i traktowane jako pojedyncza (ale nie atomowa!) instrukcja.

Wewnątrz instrukcji, kolejnością wykonania steruje również operator `,`

SEQUENCE

Control flow

```
auto x = 42;     // pierwsza instrukcja
auto y = x;      // druga instrukcja
print(y);        // trzecia instrukcja
```

SELECTION

Control flow

`if-else` wybiera następną instrukcję do wykonania na podstawie wartości logicznej dowolnego wyrażenia

`switch` wybiera nastepną instrukcję do wykonania na podstawie wartości typu wyliczeniowego[1] (*enum*)

[1] nie jest to do końca prawda – można switchować na dowolnej wartości liczbowej, ale nie jest to dobry pomysł

SELECTION (if)

Control flow

```
if (x < y) {
    std::cout << "x less than y\n";
} else if (x > y) {
    std::cout << "x greater than y\n";
} else if (x == y) {
    std::cout << "x equals y\n";
} else {
    std::cout << "something else\n";
}
```

SELECTION (`if` – OPERATORY)

Control flow

Operatory porównania (poprzedni slajd) pozwalają na zbadanie relacji między dwiema wartościami:

- a `<` b czy a jest mniejsze od b
- a `>` b czy a jest większe od b
- a `<=` b czy a jest mniejsze od lub równe b
- a `>=` b czy a jest większe od lub równe b
- a `!=` b czy a jest różne od b
- a `==` b czy a jest równe b

SELECTION (switch)

Control flow

```
switch (x) {
    case Maybe::Something:
        std::cout << "it's something\n";
        break;
    case Maybe::Nothing:
        std::cout << "it's nothing\n";
        break;
    default:
        std::cout << "it's weird\n";
        break;
}
```

ITERATION

Control flow

`while`	pętla wykonująca się dopóki wyrażenie kontrolne jest prawdziwe, ze sprawdzeniem przed wykonaniem instrukcji
`do-while`	pętla wykonująca się dopóki wyrażenie kontrolne jest prawdziwe, ze sprawdzeniem po wykonaniu instrukcji
`for`	pętla po zakresie

Pętla `while` jest najbardziej ogólną pętlą, ale wszystkie rodzaje są równoważne (każdą z pętli da się zaimplementować w ramach każdej innej).

iteration (while)

Control flow

```
while (its_sunny_outside()) {
    std::cout << "weather is nice\n";
}
```

iteration (do-while)

Control flow

```
auto x = 0;
do {
    x = roll_dice();
    std::cout << "you rolled " << x << "\n";
} while (x != 6);
```

ITERATION (for)

Control flow

```
for (auto i = 10; i >= 0; --i) {
    std::cout << i << '\n';
}
std::cout << "Happy New Year!\n";
```

PROCEDURAL ABSTRACTION

Control flow

Funkcje spełniają rolę procedur w C++. Każda funkcja składa się z:

1. nazwy – identyfikatora jakim można ją *wywołać*
2. listy parametrów formalnych – specyfikacji jakich *argumentów* (*parametrów faktycznych*) wymaga od wywołującego
3. typu zwracanego – specyfikacji typu jakiego wartości funkcja produkuje
4. ciała – ograniczonego nawiasami klamrowymi zbioru instrukcji określającego operacje jakich dana funkcja jest abstrakcją

PROCEDURAL ABSTRACTION

Control flow

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

RECURSION

Control flow

Rekurencja jest realizowana za pomocą funkcji.

RECURSION

Control flow

```
/* Raises b to the power of n. */
auto exponentiate(int const b, int const n) -> int
{
    if (n <= 0) {
        return 1;
    }
    return (b * exponentiate(b, (n - 1)));
}
```

CONCURRENCY AND PARALLELISM

Control flow

Współbieżność w C++ jest realizowana za pomocą *wątków*. Tym samym mechanizmem jest realizowana *równoległość* przetwarzania (*parallelism*).

Współbieżność można też zaimplementować na własną rękę, ale wymaga to znacznie większego nakładu pracy.

Concurrency and parallelism (std::thread)

Control flow

```
auto display_greeting(std::string const name) -> void
{
    std::cout << ("Hello, " + name + "!\n");
}

auto t1 = std::thread{display_greeting, "Joe"};     // Armstrong
auto t2 = std::thread{display_greeting, "Bjarne"}; // Stroustrup

/*
 * Threads must be joined into the parent thread, or
 * the program will crash.
 */
t1.join();  // joining thread is blocked until joined
            // thread terminates
t2.join();
```

NONDETERMINISM

Control flow

Niedeterminizm jest nieodłączną cechą równoległości – nie mamy gwarancji w jakiej kolejności będą względem siebie wykonywac sie operacje w *różnych* wątkach.

Niedeterminizm wewnątrz wątku możemy uzyskać generując liczby losowe. W tym celu można użyć `std::ranom_device` lub odczytać *n* bajtów z pliku `/dev/urandom`.

NONDETERMINISM (std::random_device)


```
std::random_device rd;
std::uniform_int_distribution<int> d20 (1, 20);

constexpr auto CRITICAL_SUCCESS = 20;
constexpr auto CRITICAL_FAILURE = 1;

auto const x = d20(rd);

if (x == CRITICAL_SUCCESS) {
    std::cout << "you kill the monster in a single blow!\n";
} else if (x == CRITICAL_FAILURE) {
    std::cout << "you wound yourself with your own sword!\n";
} else {
    std::cout << "roll for damage.\n";
}
```

EXCEPTIONS

Control flow

Mechanizmem dedykowanym sygnalizacji i obsługi błędów w C++ są *wyjątki*. Wyjątek może być rzucony (zasygnalizowany) słowem kluczowym `throw`; obsługa wyjątków odbywa się w bloku `try-catch`.

C++ pozwala na użycie dowolnego typu jako wyjątku.

EXCEPTIONS

Control flow

```
auto search_your_feelings(std::string father) -> void
{
    if (father == "Darth Vader") {
        throw std::string{"NO!!! NO!!!"};
    }
}

/* ... */

try {
    luke.search_your_feelings(lord_vader);
} catch (std::string const& error) {
    std::cerr << ("operation failed: " + error + '\n');
}
```

Overview

FUNDAMENTALNE TYPY DANYCH

Data structures

`void`	typ reprezentujący “nic”
`bool`	typ reprezentujący wartości logiczne
`char`	typ reprezentujący znaki
`int`	typ reprezentujący liczby całkowite
`double`	typ reprezentujący liczby zmiennoprzecinkowe

`https://en.cppreference.com/w/cpp/language/types`

void

FUNDAMENTALNE TYPY DANYCH

Typ `void` jest często używany jeśli funkcja nie produkuje żadnej wartości[2]:

```
auto produce_nothing() -> void
{}
```

[2] takie funkcje nazywane są czasem "procedurami", ang. *procedure*

bool

FUNDAMENTALNE TYPY DANYCH

Typ `bool` jest zwracany przez operatory porównania (slajd 9). Jest również typem "argumentu" instrukcji `if`. Przyjmuje jedynie dwie wartości – reprezentowane słowami kluczowymi `true` i `false`:

```
auto you_know_it_to_be = true;
auto lies_and_deception = false;
```

char

FUNDAMENTALNE TYPY DANYCH

Typ `char` jest używany do reprezentowania znaków. Wywodzi się z czasów kiedy królowało kodowanie ASCII, a znaki (oraz kody kontrolne) zajmowały sztywne 7 bitów.

```
auto a = 'A';
auto new_line = '\n';
```

Domyślnie typ `char` reprezentuje wartości "ze znakiem" czyli de facto `signed char`; możliwe jest wymuszenie "bezznakowości" pisząc `unsigned char`.

int

FUNDAMENTALNE TYPY DANYCH

Typ `int` jest używany do reprezentowania liczb całkowitych, domyślnie ze znakiem:

```
auto answer_to_everything  = 42;
auto negative_of_the_beast = -666;
```

Typ `int` można "doprecyzować" specyfikatowami znaku i rozmiaru:

`signed` liczba całkowita ze znakiem (domyślnie)

`unsigned` liczba całkowita bez znaku

`short` "krótka" liczba, czyli zazwyczaj 16 bitów

`long` "długa" liczba, czyli zazwyczaj 64 bity

Specyfikatory można łączyć, np. w tym `unsigned long int`.

double

FUNDAMENTALNE TYPY DANYCH

Typ `double` jest używany do reprezentowania liczb zmiennoprzecinkowych. Jego mniej precyzyjną alternatywą jest typ `float`.

```
auto pi      = 3.14f;    // float
auto piiiii = 3.14159; // double
```

Domyślnym typem literałów zmiennoprzecinkowych jest `double`.

ZŁOŻONE TYPY DANYCH

Data structures

Złożone (ang. *compound*) typy danych składają się z typu podstawowego i "dodatku":

`T[n]`	tablica n wartości typu `T`
`T*`	wskaźnik do obszaru pamięci zawierającego wartość typu `T`
`T&`	referencja do wartości typu `T`
`T const`	stała wartość typu `T`

`https://en.cppreference.com/w/cpp/language/type`

TABLICA

ZŁOŻONE TYPY DANYCH

Tablice, odziedziczone z języka C, służą do przechowywania kilku wartości tego samego typu:

```
int decimal_digits[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

Tablica może mieć również rozmiar nieokreślony – widać to np. w funkcji `main()`:

```
auto main(int argc, char* argv[]) -> int
{
    return 0;
}
```

WSKAŹNIK I REFERENCJA

ZŁOŻONE TYPY DANYCH

Wskaźniki, odziedziczone z języka C, służą do przechowywania adresów obszarów pamięci zawierających wartości typu T:

```
auto a_value   = 42; // or, explicitly: int a_value ...
auto a_pointer = &a; // or, explicitly: int* a_pointer ...
```

Referencje służą do przechowywania odniesień do wartości typu T:

```
auto  a_value = 42; // or, explicitly: int a_value ...
auto& a_ref   = a;  // or, explicitly: int& a_pointer ...
```

Wskaźniki mogą być "zerowe" tzw. *null pointer* i nie wskazywać na poprawny adres. Referencje *zawsze* wskazują na istniejącą wartość[3].

[3] przynajmniej w teorii...

BIBLIOTEKA STANDARDOWA

Data structures

Biblioteka standardowa (ang. *standard library*) języka C++ zawiera wiele struktur danych takich jak:

`std::array`	"silna" tablica
`std::vector`	tablica o zmiennej długości
`std::queue`	kolejka FIFO
`std::map`	struktura mapująca klucze typu T na wartości typu V
`std::string`	napis
`std::pair`	para wartości typów F i S

Warto używać struktur z biblioteki standardowej żeby oszczędzić sobie pracy.

WŁASNE TYPY DANYCH

Data structures

Programista C++ może definiować również własne typy danych: struktury i klasy, oraz wyliczenia.

Klasy (`class`) różnią się od struktur (`struct`) tylko i wyłącznie tym, że ich pola są domyślnie publiczne.

Wyliczenia słabe (`enum`) są typu `int`, ich wartości są globalne, i mają automatycznie zdefiniowane operacje arytmetyczne (np. sumę bitową). Są przydatne przy definiowaniu flag, które można łączyć.
Wyliczenia silne (`enum class`) różnią sie od słabych tym, że są "swojego własnego" typu, ich wartości nie są globalne, oraz nie mają automatycznie zdefiniowanych operacji arytmetycznych. Są przydatne przy definiowaniu rozdzielnych stanów.

STRUKTURY (struct)

Data structures

```
struct being_with_legs {
    std::string const name;
    size_t const legs;

    being_with_legs(std::string, size_t);
};

being_with_legs::being_with_legs(std::string n, size_t l)
    : name{std::move(n)}
    , legs{l}
{}

/* ... */

auto const snake = being_with_legs{ "snake", 0 };
auto const human = being_with_legs{ "human", 2 };
auto const spider = being_with_legs{ "spider", 8 };
```

wyliczenia "silne" (enum class)

Data structures

```
enum class meal_kind {
    BREAKFAST,
    DINNER,
    SUPPER,
};

auto is_most_important_meal_of_the_day(meal_kind const meal) -> bool
{
    return (meal == meal_kind::BREAKFAST);
}
```

wyliczenia (enum)

Data structures

```
enum some_flags_type {
    SOME_FLAG_READ,
    SOME_FLAG_WRITE,
    SOME_FLAG_NONBLOCK,
    SOME_FLAG_BUFFERED,
    SOME_FLAG_UNBUFFERED,
};
constexpr auto SOME_FLAG_DEFAULT = SOME_FLAG_READ
                                 | SOME_FLAG_WRITE
                                 | SOME_FLAG_BUFFERED;

// We want read-only, non-blocking, unbuffered descriptor.
auto const mode = SOME_FLAG_READ
                | SOME_FLAG_NONBLOCK
                | SOME_FLAG_UNBUFFERED;
```

Overview

I/O

C++ korzysta z mechanizmów I/O dostarczanych przez API systemu operacyjnego (np. Linux), ale część z nich opakowuje w swoje własne abstrakcje zapewniając programom przenośność.

STANDARD STEAMS

I/O

W momencie uruchomienia dla większości programów tworzone są 3 standardowe strumienie: wejścia, wyjścia, i błędów.

`std::cin` standardowy strumień wejścia, służy do odczytu danych podawanych przez użytkownika w konsoli tekstowej (*file descriptor 0*)

`std::cout` standardowy strumień wyjścia, służy do prezentacji wyników działania programu w konsoli tekstowej (*file descriptor 1*)

`std::cerr` standardowy strumień błędów, służy do prezentacji błędów działania i awarii programu w konsoli tekstowej (*file descriptor 2*)

STANDARD STEAMS

I/O

```
{
    std::string line;

    // read a line of text from standard input
    std::getline(std::cin, line);
}

// display a message to inform user of what is happening...
std::cerr << "connecting to server...\n";

// ...or notify them about errors
std::cerr << "connection failed\n";

// display results of program's work
std::cout << downloaded_data << '\n';
```

FILES

I/O

C++ definiuje typy `std::ifstream` (*input file stream*) i `std::ofstream` (*output file stream*) w bibliotece standardowej.

Jeśli ich interfejs nie jest wystarczający zawsze można użyć interfejsu platformy czyli np. wywołań systemowych definiowanych przez standard POSIX – `open(3)`, `write(3)`, `read(3)`, i `close(3)`.

files (std::ifstream and std::ofstream)


```
auto path = std::string{"./data.txt"};

{
    // write line to a file
    auto out = std::ofstream{ path };
    if (out.good()) {
        out << "Hello, World!\n";
    }
}

{
    // read line from a file
    auto in = std::ifstream{ path };
    if (in.good()) {
        auto line = std::string{};
        std::getline(in, line);
        std::cout << line << "\n";
    }
}
```