Podstawy programownia (w języku C++)

Wstęp do programowania

Marek Marecki

Polsko-Japońska Akademia Technik Komputerowych

4 listopada 2020

Overview

Środek historii

...czyli 150 lat do CPU

1822 Difference Engine[1] – Charles Babbage

1837 Analytical Engine – Charles Babbage hardware[2] [3], a Ada Lovelace software

1941 Konrad Zuse – Z3, pierwszy programowalny komputer[4]

1944 Harvard Mark I – drugi programowalny komputer

1971 Intel 4004 – 4-bitowy mikroprocesor

W 2016 UK pozwoliło sprzedać ARM – czyli po raz drugi wypuścili z rąk ważny kawałek technologii.

[1] skonstruowany w londyńskim Science Museum w 1991

[2] zbudował prototyp CPU w 1871

[3] w 1906 jego syn, Henry Babbage, zbuduje kompletene CPU

[4] zniszczony podczas bombardowania Berlina przez Aliantów

Architektury CPU

1837 Analytical Engine

1978 x86

1985 ARM, MIPS

1991 PowerPC

2001 Itanium (VLIW; *failed*)

2003 Mill (VLIW; *in development*)

2010 RISC-V

Overview

von Neumann

1. CPU
2. RAM
3. pamięć masowa
4. I/O

Overview

A co to?

Sposób na wyrażenie swoich żądań względem maszyny.

Kontrakt z demonem – spełnia rozkazy *dokładnie tak jak są wypowiedziane*, bez oglądania się na *intencje* programisty.

A po co to komu?

```
mov eax, 0x2a (x86)
```
vs
```
auto x = int{42}; (C++)
```

Dużo prostsze wydawanie maszynie skomplikowanych rozkazów, i łatwość zrozumienia znaczenia programu.
Automatyczna alokacja rejestrów i pamięci; automatyczne skoki; ergonomiczna semantyka.

Przenośność (ang. *portability*) programów między platformami.

Umowny podział

1. compiled *vs* interpreted (JIT?)
2. typing: static *vs* dynamic, strong *vs* weak
3. paradigm[5]: functional *vs* object-oriented *vs* structural *vs* prototype-based *vs* ...
4. rodziny: C-like (pochodne po języku ALGOL), ML-like, Lisp-like
5. "toy" *vs* "real"

C	compiled, static-weak typing, structural
C++	compiled, static-strong typing, multiparadigm
Smalltalk	interpreted, dynamic-strong, object-oriented
OCaml	compiled, static-strong, functional
Perl	interpreted, dynamic-weak, multiparadigm

[5] Seven Languages in Seven Weeks; Bruce A. Tate; ISBN-13: 978-1-934356-59-3

Overview

Co jest potrzebne w języku?

? (pytanie do sali)

Z punktu widzenia prostego człowieka

1. *control flow* – mechanizmy przepływu kontroli, czyli sterowania programem
2. *data structures* – reprezentacja struktur danych
3. *I/O* – zapis i odczyt danych, czyli sposób na interakcję ze światem zewnętrznym

Jackson Structured Programming

Control flow

Michael Jackson, 1975; Principles of Program Design

1. *sequence* – sekwencjonowanie, czyli ustalenie kolejności wykonywania operacji
2. *selection* (*alternative*) – wybór (alternatywa), czyli decyzja o podjęciu jednej z kilku różnych ścieżek wykonania
3. *iteration* – iteracja, czyli powtarzanie tych samych kroków n razy

Nadaje się do opisu algorytmów, ale nie za bardzo do czegoś więcej. Często tak jest z różnymi modelami – są wygodne w teorii, ale niezbyt praktyczne.

Warnier/Orr

Control flow

Jean-Dominique Warnier, 1976; Logical construction of programs
Kenneth Orr, 1977; Structured systems development

1. *recursion* – rekurencja, czyli sposób na zagnieżdżone wykonywanie operacji
2. *concurrency* – współbieżność, czyli sposób na wykonywanie kilku operacji "w tym samym czasie" (naprzemiennie na jednym procesorze, lub równolegle[6] na wielu)

Rekurencja i współbieżność są nieodłącznymi elementami programów, które działają w "prawdziwym świecie". Bez nich niemożliwe byłoby interaktywne używanie komputerów.

[6] ten wariant nazywa się *parallelism*, i czasem jest podawany obok współbieżności jako coś innego

Michael Scott

Control flow

Michael Lee Scott, 2000; Programming language pragmatics[7]

1. *procedural abstraction* – zbiór operacji opakowany w sposób umożliwiający ich wspólne wywołanie, w skrócie: funkcja
2. *nondeterminacy* – niedeterminizm, czyli sposób na zapewnienie losowości przy wyborze ściezki wykonania
3. *exceptions** – wyjątki, pozwalające na ”skok” kontroli w przypadku wystąpienia błędu

Funkcje i niedeterminizm zamykają bazowe mechanizmy, które służą kontroli przepływu.

[7] ISBN 1-55860-442-1

Podsumowanie

Control flow

1. *sequence*
2. *selection*
3. *iteration*
4. *recursion*
5. *concurrency*
6. *procedural abstraction*
7. *nondeterminism*
8. *exceptions**

Egzotyczne metody kontroli przepływu – *continuations*, *coroutines*.

Bit, nibble[8], byte, word, half-word, double-word, quad-word...

Data structures

Na początku było słowo

J 1,1-3

Bit - czyli wartość mogąca przechowywać 0 lub 1.
Podstawową jednostką danych obsługiwanych przez CPU jest ”słowo” - sekwencja bitów o pojedynczego długości rejestru. Dla architektury x86-64 długość słowa to 64 bity.
Zapis i odczyt słowa w pamięci zazwyczaj są *operacjami atomowymi* co ma znaczenie dla programowania współbieżnego.

[8] połowa bajtu, czyli 4 bity

Liczby całkowite, ułamki, wartości logiczne

Data structures

Liczby całkowite - ze znakiem (`signed`), bez znaku (`unsigned`).
Liczby zmiennoprzecinkowe - pojedynczej precyzji, podwójnej precyzji.
Wartości logiczne - prawda, fałsz.

Znaki, napisy

Data structures

Znaki[9] - reprezentujące pojedynczy glif (literę, znak interpunkcyjny, itd.) lub symbol kontrolny.
Napisy - reprezentujące sekwencje znaków (np. ''`Hello, World!`'').

[9] kiedyś były najczęśniej szerokości 1 bajty (ASCII), ale obecnie, od upowszechnienia się standardu Unicode, są zazwyczaj zmiennej długości (UTF-8)

Lista, kolejka, stos, zbiór, drzewo, krotka...

Data structures

1. *list* – lista, czyli poszeregowana sekwencja wartości typu T, do których daje dostęp w dowolnej kolejności (ang. *random access*)
2. *queue* – kolejka, czyli poszeregowana sekwencja wartości typu T, do których daje dostęp na zasadzie *FIFO*
3. *stack* – stos, czyli poszeregowana sekwencja wartości typu T, do których daje dostęp na zasadzie *LIFO*
4. *set* – zbiór, nieposzeregowana kolekcja wartości typu T
5. *tree* – drzewo, często wykorzystywane do budowania "map" czyli struktur asocjacyjnych pozwalających na przechowanie wartości typu T pod kluczem typu K
6. *tuple* – krotka, czyli struktura danych zawierająca n pól typów T_0, T_1, ..., T_n

Podstawową złożoną strukturą danych jest *tablica*, czyli sekwencja n elementów typu T. Bazując na tablicach bajtów (czyli de facto surowych, wydzielonych obszarach pamięci) można zaimplementować wszystkie powyższe struktury danych.

Typy użytkownika

Data structures

enum wyliczenie, czyli zdefiniowany zbiór wartości, które dany typ może przechować (vide *sum type*)

struct struktura, czyli typ złożony z kilku pól różnych typów – może przechowywać wszystkie kombinacje wartości pól (vide *product type*)

Języki programowania często zapewniają programistom możliwość tworzenia własnych typów danych.

Wskaźniki i dynamiczna alokacja pamięci

Data structures

Implementacja wielu struktur danych (np. list lub napisów o zmiennej długości) byłaby niemożliwa bez wskaźników i dynamicznej alokacji pamięci.

Podsumowanie

Data structures

1. słowo (*word*), bajt (*byte*), bit
2. tablica (*array*)
3. typy użytkownika (*user-defined type – enum, struct*)
4. wskaźnik (*pointer*)
5. dynamiczna alokacja pamięci (*memory allocation*)
6. typy proste *vs* typy złożone

Operacje wejścia-wyjścia

I/O

Sposób na interakcję i wymianę danych ze "światem zewnętrznym", czyli wszystkim tym co dzieje się poza CPU i pamięcią operacyjną (RAM).

1. *I/O port*, *MMU* (ang. *memory management unit*)
2. *file-descriptor*, *socket* (ten sam interfejs dla plików i połączeń w sieci; POSIX)
3. *memory-mapped file*

Zapisując i odczytując bajty da się obsłużyć każdy rodzaj urządzenia - monitor, klawiaturę, dysk, ramię robota, silnik, itd.

Overview

Składniki języka 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.

Control flow - sequence

Składniki języka C++

Kontrola w języku C++ zaczyna się od pierwszej instrukcji w funkcji `main()`. Kolejne instrukcje wykonywane są w kolejności zdefiniowanej w kodzie źródłowym i oddzielone operatorem `;`

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

Wewnątrz instrukcji, kolejnością wykonania streuje również operator `,`

Control flow - sequence

Składniki języka C++

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

Control flow - selection

Składniki języka C++

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

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

Control flow - selection (if)

Składniki języka C++

```
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";
}
```

Control flow - selection (switch)

Składniki języka C++

```
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;
}
```

Control flow - iteration

Składniki języka C++

`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).

Control flow - iteration (while)

Składniki języka C++

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

Control flow - iteration (do-while)


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

Control flow - iteration (for)

Składniki języka C++

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

Control flow - procedural abstraction

Składniki języka C++

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

Control flow - procedural abstraction

Składniki języka C++

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

Control flow - recursion

Składniki języka C++

Rekurencja jest realizowana za pomocą funkcji.

Control flow - recursion

Składniki języka C++

```
/* 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)));
}
```

Control flow - concurrency and parallelism

Składniki języka C++

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.

Control flow - concurrency and parallelism (std::thread)

Składniki języka C++

```
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();
```

Control flow - nondeterminism

Składniki języka C++

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`.

Control flow - 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";
}
```

Control flow - exceptions

Składniki języka C++

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.

Control flow - exceptions

Składniki języka C++

```
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');
}
```

Data structures - biblioteka standardowa

Składniki języka C++

Biblioteka standardowa (ang. *standard library*) języka C++ zawiera wiele struktur danych takich jak `std::vector` (sekwencja o zmiennej długości), `std::queue` (kolejka FIFO), `std::map` (struktura mapująca klucze na wartości), `std::string` (napis), `std::pair`, itd.

Warto używać struktur (a także funkcji) z biblioteki standardowej żeby oszczędzić sobie pracy.

Data structures - własne typy danych

Składniki języka C++

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.

Data structures - struktury (struct)

Składniki języka C++

```
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 };
```

Data structures - wyliczenia (enum class)

Składniki języka C++

```
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);
}
```

Data structures - wyliczenia (enum)

Składniki języka C++

```
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;
```

I/O

Składniki języka C++

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.

I/O - standard steams

Składniki języka C++

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*)

I/O - standard steams

Składniki języka C++

```
{
    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';
```

I/O - files

Składniki języka C++

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, np. wywołań systemowych definiowanych przez standard POSIX – `open(3)`, `write(3)`, `read(3)`, i `close(3)`.

I/O - 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";
    }
}
```

I/O - sieć

Składniki języka C++

Komunikacja po sieci odbywa się z wykorzystaniem mechanizmów I/O dostarczanych przez standard POSIX – `socket(2)`, `bind(2)`, `listen(2)`, `accept(2)`, `connect(2)`, `inet_pton(3)`, `connect(3)`, `shutdown(3)`, `close(3)`.

I/O - sieć (nagłówki)

Składniki języka C++

Pliki nagłówkowe, które należy dodać na początku pliku z kodem źródłowym żeby móc bez przeszkód korzystać z funkcji pozwalających na komunikację po sieci:

```
#include <arpa/inet.h>
#include <endian.h>
#include <netinet/ip.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
```

I/O - sieć (klient)

Składniki języka C++

```
auto sock = socket(AF_INET, SOCK_STREAM, 0);

auto const ip = std::string{"127.0.0.1"};
auto const port = uint16_t{42420};

sockaddr_in server;
memset(&server, 0, sizeof(server));
server.sin_family = AF_INET;
server.sin_port = htobe16(port);
inet_pton(AF_INET, ip.c_str(), reinterpret_cast<void*>(&server.sin_addr));

connect(sock, reinterpret_cast<sockaddr*>(&server), sizeof(server));

auto const data = std::string{"Hello, World!"};
write(sock, data.c_str(), data.size());

shutdown(sock, SHUT_RDWR);
close(sock);
```

I/O - sieć (serwer, część $\frac{1}{2}$)

Składniki języka C++

```
auto sock = socket(AF_INET, SOCK_STREAM, 0);

auto const ip = std::string{"127.0.0.1"};
auto const port = uint16_t{42420};

sockaddr_in server;
memset(&server, 0, sizeof(server));

server.sin_family = AF_INET;
server.sin_port = htobe16(port);

inet_pton(AF_INET, ip.c_str(), reinterpret_cast<void*>(&server.sin_addr));

bind(sock, reinterpret_cast<sockaddr*>(&server), sizeof(server));
listen(sock, 0);  // default limit of waiting connections
```

W ten sposób przygotuje się socket do odbierania połączeń.

I/O - sieć (serwer, część $\frac{2}{2}$)

Składniki języka C++

```
sockaddr_in client_addr;
memset(&client_addr, 0, sizeof(client_addr));
auto client_len = socklen_t{sizeof(client_addr)};
auto client = accept(sock,
    reinterpret_cast<sockaddr*>(&client_addr), &client_len);
{
    // report client's address
    std::array<char, INET_ADDRSTRLEN + 1> buffer {};
    inet_ntop(AF_INET, &client_addr.sin_addr, buffer.data(), buffer.size());
    std::cout << buffer.data() << ':' << be16toh(client_addr.sin_port) << "\n";
}
{
    // read data from client
    std::array<char, 512> buffer {};
    auto const n = read(client, buffer.data(), buffer.size());
    std::cout << std::string{buffer.data(), buffer.data() + n} << "\n";
}

shutdown(sock, SHUT_RDWR);
close(sock);
```

W ten sposób odbiera się połączenia i rejestruje adres połączonego klienta.