Объектно-ориентированное программирование

Object-oriented programming

XI. Ввод и вывод

Input/output

Принцип отложенного (deferred) вычисления

Вычисление результата выражения откладывается до того момента, когда значение выражения становится необходимым.

- Каждый элемент коллекции рассматривается отдельно* (lazy evaluation)
- Элемент коллекции рассматривается вместе** с остальными (eager evaluation)

https://learn.microsoft.com/en-us/dotnet/standard/linq/deferred-execution-lazy-evaluation

Потоки (streams*) в UNIX

Дуплекс-соединение между процессом и “устройством”.

- Обработчик потока организован как **стек**
- Обработчик потока состоит из двух **очередей**
- Очереди обмениваются “блоками”, сост. из read-, write- и limit-**указателей**
- Активация очереди вызывает удаление блоков с данными и помещает их в следующую очередь

https://cseweb.ucsd.edu/classes/fa01/cse221/papers/ritchie-stream-io-belllabs84.pdf

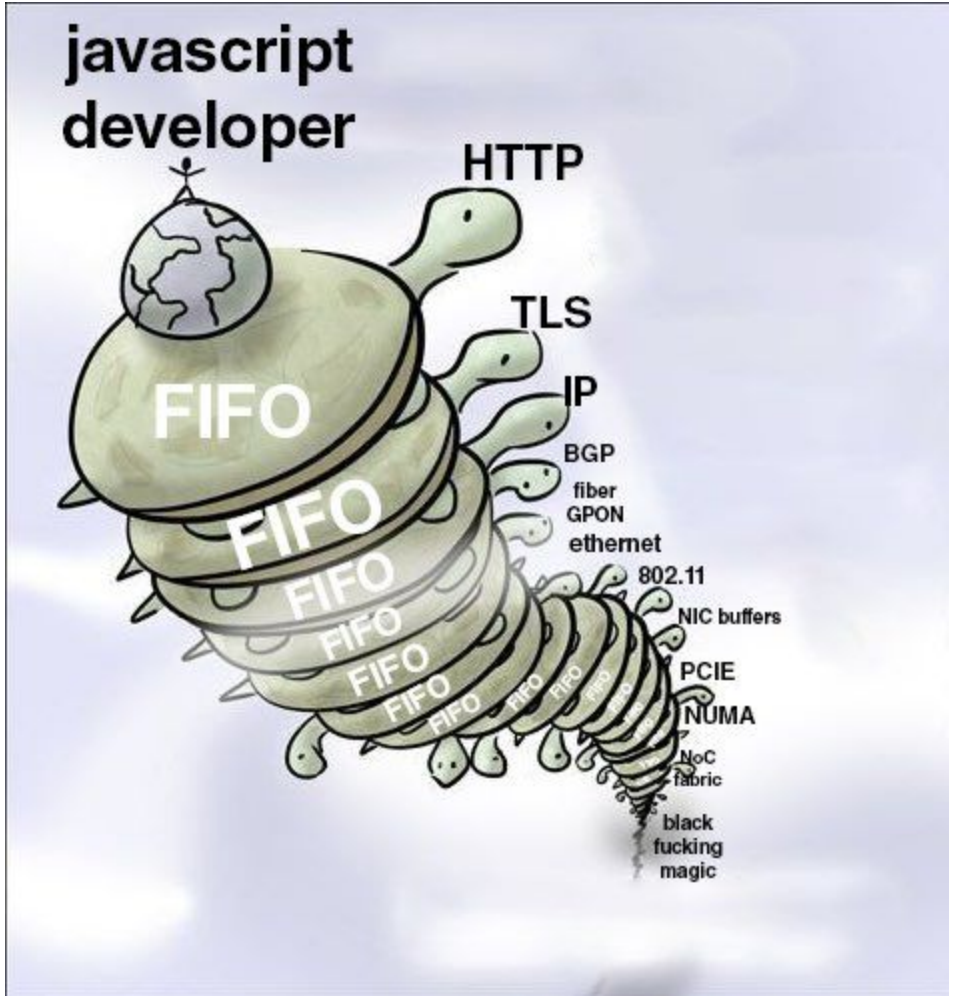
Устройство очереди сообщений в UNIX

```
struct queue {
    int flag;             /* биты для разных флагов */
    void (*putp)();       /* процедура отправки сообщения */
    void (*servp)();      /* процедура приема сообщения */
    struct queue *next;   /* указатель на следующую очередь в потоке */
    struct block *first; /* первое сообщение */
    struct block *last;   /* последнее сообщение */
    int hiwater;          /* максимальная вместимость */
    int lowater;          /* сигнал для принятия новых сообщений */
    int count;            /* текущие символы */
    void *ptr;            /* скрытая область для хранения данных */
};
```

Примеры использования

- Каналы* stdin, stdout, stderr (channels)
- Конвейеры (pipes, inter-process communication)
- Генераторы** и со-программы (yield)
- Примитивы синхронизации (future vs. promise)

и т.п.


javascript
developer
HTTP
TLS
IP
BGP
fiber
GPON
ethernet
802.11
NIC buffers
PCIE
NUMA
NoC
fabric
black
fucking
magic
FIFO
FIFO
FIFO
FIFO
FIFO
FIFO

Каналы* (channels) в UNIX

stdio.h:

```
_CRTIMP FILE *__cdecl __acrt_iob_func(unsigned index);

#define stdin  (__acrt_iob_func(0))
#define stdout (__acrt_iob_func(1))
#define stderr (__acrt_iob_func(2))
```

```
fclose(stdout);
stdout = fopen("/standard/output.txt", "w");
```

https://web.archive.org/web/20230127013534/http://www.di.uevora.pt/~lmr/syscalls.html

XII. Конвейеры

Pipelines

Анекдот

Read a file of text, determine the n most frequently used words, and print out a sorted list of those words along with their frequencies.

Прочитать текстовый файл, определить **n** самых часто встречающихся слов, вывести на экран отсортированный список этих слов вместе с их частотой повторения.

https://leancrew.com/all-this/2011/12/more-shell-less-egg/

```
tr -cs A-Za-z '\n' |
tr A-Z a-z         |
sort               |
uniq -c            |
sort -rn           |
sed ${1}q
```

“A wise engineering solution would produce — or better, exploit — **reusable parts**.”

M. “Doug” McIlroy

https://dl.acm.org/doi/pdf/10.1145/5948.315654

Конвейеры (UNIX pipelines)

"We should have some ways of **coupling programs** like garden hose – screw in another segment when it becomes necessary to massage data in another way. **This is the way of IO** also."

M. "Doug" McIlroy

"The Origin of Unix Pipes", October, 11, 1964*

https://dsf.berkeley.edu/cs262/unix.pdf

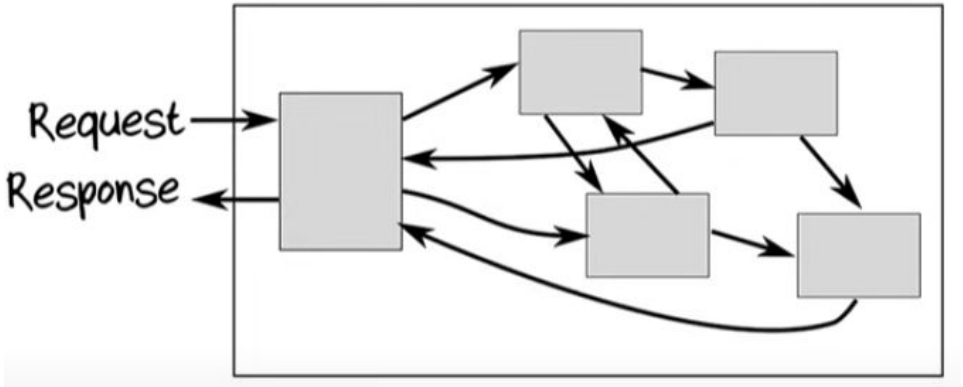
```
{
    "updated": "2022-01-10T00:35:00Z",
    "data": [
        {
            "Code": "111111",
            "Occupation": "Chief Executive or Managing Director",
            "Count": 54480
        },
...
        {
            "Code": "999999",
            "Occupation": "Not stated",
            "Count": 0
        }
    ]
}
```

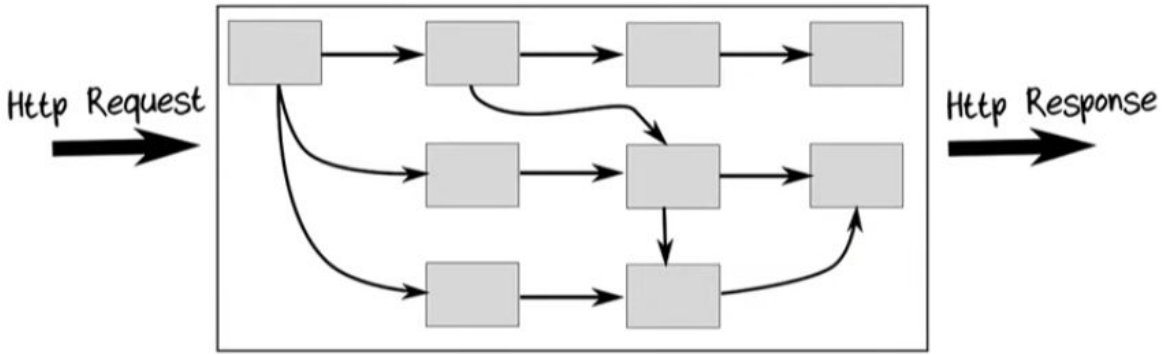
Пример использования конвейера

```
curl -XGET https://my-file-location.io/stats.json > tmp1
grep “\”Count\”: 0” < tmp1 > tmp2
wc -l < tmp2
```

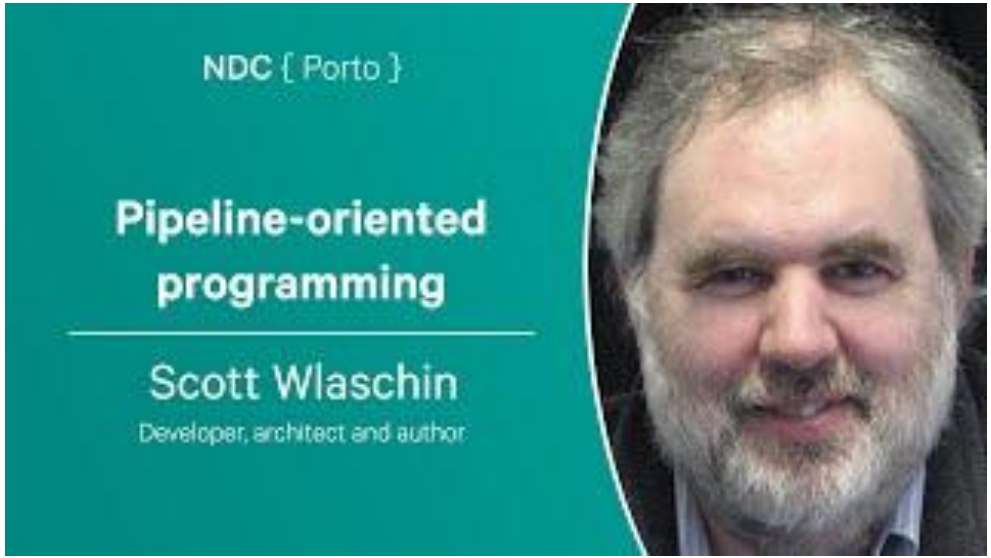
```
curl -XGET http://my-file-location.io/stats.json
| grep “\”Count\”: 0”
| wc -l
```

Конвейеры против объектов





https://www.youtube.com/watch?v=ipceTuJlw-M



"Everything there—even input conversion and sorting—is programmed monolithically and from scratch. In particular the isolation of words, the handling of punctuation, and the treatment of case distinctions are built in. Even if data-filtering programs for these exact purposes were not at hand, these operations would well be implemented separately: for **separation of concerns**, for easier development, for **piecewise debugging**, and for **potential reuse**."

M. "Doug" McIlroy

https://dl.acm.org/doi/pdf/10.1145/5948.315654

XIII. Генераторы

Generators

Thunks*

Процессы, которые первоначально использовались для параллельного вычисления значений выражений для параметров функции при вызове.

“A piece of coding which provides an address”

P. Z. Ingerman

1961

Thunks

В отличие от **callback**-функций, которые явл. адресом на код:

```
void qsort(void *ptr, size_t count, size_t size,
           int (*comp)(const void*, const void*) );
```

Пример **thunk**-ов в Scheme:

```
(define ones
        (lambda () (cons 1 ones)))
```

```
(define (natural x)
        (cons x (lambda () (natural (+ x 1)))))
```

Closures

[...]a **data structure** containing a **lambda expression**, and an **environment** to be used when that lambda expression is applied to arguments

https://en.wikisource.org/wiki/Page:Scheme_-_An_interpreter_for_extended_lambda_calculus.djvu/22

Closures

```
def f(x):
    def g(y):
        return x + y
    return g

def h(x):
    return lambda y: x + y

a = f(1)
b = h(1)

assert a(5) ==    ?
assert b(5) ==    ?
assert f(1)(5) == ?
assert h(1)(5) == ?
```

Closures

```
def f(x):
    def g(y):
        return x + y
    return g

def h(x):
    return lambda y: x + y

a = f(1)
b = h(1)

assert a(5) ==    6
assert b(5) ==    6
assert f(1)(5) == 6
assert h(1)(5) == 6
```

Замыкание (closure)

Закрытое выражение – структура данных, состоящая из **λ-выражения** и **среды***, относительно которой выражение вычисляется, т. е. это:

- список из двух сущностей: среды и идентификатора/списка идентификаторов;
- единичный список из аппликативного выражения.

Для выражения `X = λb.B`, ее связанных переменных `bv`, среды `E` замыкание определено как:

```
cons((E, bv), unitList(B)).
```

https://www.cs.cmu.edu/~crary/819-f09/Landin64.pdf

Лямбда-функции в C++

"Constructs a **closure** (an unnamed function object capable of capturing variables in scope)."

[captures] front-attr (optional)

(params) specs (optional) exception (optional)
back-attr (optional) trailing-type (optional) requires (optional)

{ body }

https://en.cppreference.com/w/cpp/language/lambda

Лямбда-функции в C++

[captures]

a comma-separated list of zero or more captures, optionally beginning with the **capture-default**. The capture list defines the **outside variables** that are **accessible from** within the lambda function **body**.

(params)

{ body }

Лямбда-функции в C++

```
void f(int i) {
    [&] {};           // OK: by-reference capture default
    [&, i] {};        // OK: by-reference capture, except i is captured by copy
    [&, this] {};     // OK, equivalent to [&]
    [&, this, i] {}; // OK, equivalent to [&, i]
}
```

```
void f(int i) {
    [=] {};          // OK: by-copy capture default
    [=, &i] {};      // OK: by-copy capture, except i is captured by reference
    [=, *this] {}; // OK: captures the enclosing S2 by copy
    [=, this] {};  // OK: same as [=]
}
```

Лямбда-функции в C++

[captures]	a comma-separated list of zero or more captures, optionally beginning with the **capture-default**. The capture list defines the **outside variables** that are **accessible from** within the lambda function **body**.
(params)	**parameter list of operator()** of the closure type.

{ body }

Лямбда-функции в C++

```
auto make_function(int& x) {
    return [&] { std::cout << x << '\n'; };
}
```

```
    int i = 3;
    auto f = make_function(i); // x связывается с i
    i = 5;
    f(); // ???
```

Лямбда-функции в C++

```
auto make_function(int& x) {
    return [&] { std::cout << x << '\n'; };
}
```

```
    int i = 3;
    auto f = make_function(i); // x связывается с i
    i = 5;
    f(); // 5
```

Еще примеры

```
std::vector<int> c{1, 2, 3, 4, 5, 6, 7};
int x = 5;
c.erase(
    std::remove_if(
        c.begin(), c.end(), [x](int n) { return n < x; }), c.end());
// ???
```

```
auto nth_fibonacci = [](int n) {
    std::function<int(int, int, int)> fib = [&](int n, int a, int b) {
        return n ? fib(n - 1, a + b, a) : b;
    };
    return fib(n, 0, 1);
};
```

Еще примеры

```
std::vector<int> c{1, 2, 3, 4, 5, 6, 7};
int x = 5;
c.erase(
    std::remove_if(
        c.begin(), c.end(), [x](int n) { return n < x; }), c.end());
// 5, 6, 7
```

```
auto nth_fibonacci = [](int n) {
    std::function<int(int, int, int)> fib = [&](int n, int a, int b) {
        return n ? fib(n - 1, a + b, a) : b;
    };
    return fib(n, 0, 1);
};
```

Генераторы в C++

```
struct pair {
    int car;
    struct pair (*cdr)();
};

auto natural(int x) -> pair {
    auto f = [=]() -> pair {
        return natural(x + 1);
    };
    return pair{x, f};
}
```

```
auto res = natural(1);
for(int i = 1; i <= 10; res = res.cdr(), i = res.car) {
    std::cout << i << " ";
}
```

Генераторы в Lisp

```
(require 'generator)
(iter-defun my-iter (x)
  (iter-yield (1 + (iter-yield (1 + x))))
  -1)

(let* ((iter (my-iter 5))
    (print (iter-next iter)) )
```

https://www.gnu.org/software/emacs/manual/html_node/elisp/Generators.html

Генераторы** в Python

```
def natural(x):
    while(True):
        yield x
        x += 1

for i in natural(1):
    if(i > 10):
        break
    print(i, end=" ")
```

https://wiki.python.org/moin/Generators

XIV. Будущности и перспективы

Futures and promises

Future - это абстракция времени

“The construct (future X) immediately returns a **future** for the value of the expression X and **concurrently** begins evaluating X. When the evaluation of X **yields** a value, that value replaces the future.”

Halstead, 1985

Future vs. promise

“Consider an "**eager** beaver" **evaluator** for an applicative programming language which starts evaluating every subexpression as soon as possible, and in parallel. This is done through the mechanism of ***futures****, which are roughly Algol-60 "**thunks**" which have their own evaluator process ("thinks"?).”

H. Baker, Jr., C. Hewitt

“The Incremental Garbage Collection of Processes”, 1977

https://web.archive.org/web/20200621015121/http://home.pipeline.com/~hbaker1/Futures.html

Promise - это процесс, вычисляющий future

“If you **promise** something to someone, you are responsible for keeping it, but if someone else makes a promise to you, you expect them to honor it in the **future**.”

http://dist-prog-book.com/chapter/2/futures.html

“Будущности” и “перспективы”

Future состоит из:

- вычислительного процесса (с отдельной **средой**);
- ячейки памяти (для кэширования значения аргумента);
- очереди процессов, ожидающих результат вычисления.

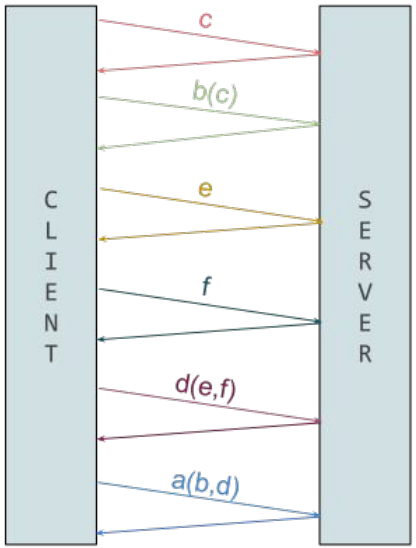
```
tmp1 := x.f();
tmp2 := y.g();
tmp3 := tmp1.h(tmp2); // tmp3 := x.f().h(y.g());
```

```
tmp3 := x <- f() <- h( y <- g() );
```

Eventuals

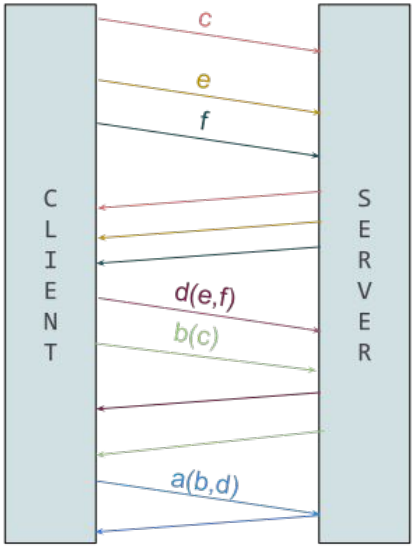
Conventional RPC

```
exp = a ( b ( c () ), d ( e (), f () ) )
```

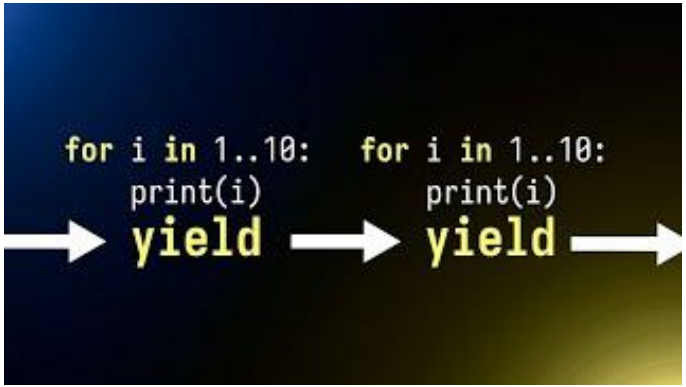


With Pipelining

```
exp = a ( b ( c () ), d ( e (), f () ) )
```



https://www.youtube.com/watch?v=sYSP_elDdZw



https://www.youtube.com/watch?v=IGv5WYYmyfo

Future и promise в C++

```
#include <thread>
#include <iostream>
#include <future>

int main() {
    std::promise<int> p;
    std::future<int> f = p.get_future();
    std::thread t([&p]{
        try { /* code that may throw */ } catch(...) {
            p.set_exception(std::current_exception());
            // store anything thrown in the promise
        }
    });
    try { std::cout << f.get(); } catch(const std::exception& e) {}
    t.join();
}
```

https://en.cppreference.com/w/cpp/thread/future

std::future

```
01: // sender/producer
02: std::future<int> async_algo() {
03:     std::promise<int> p;
04:     auto f = p.get_future();
05:     std::thread t { [p = std::move(p)] () mutable {
06:         int answer = // compute!
07:         p.set_value(answer);
08:     }};
09:     t.detach();
10:     return f;
11: }
```

https://www.usenix.org/system/files/conference/osdi16/osdi16-guerraoui.pdf