# Объектно-ориентированное программирование

Object-oriented programming

## V. Потоки данных

Streams

# Принцип отложенного 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

# Примеры использования

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

и т.п.

# Каналы (UNIX channels)

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

# 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*

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

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

# Пример использования конвейера

```
{
    "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 без использования конвейера

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

# 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

# Thunks

## Пример в C:

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

## Пример в Scheme:

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

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

# Замыкание (closure)

***Закрытое выражение*** – набор данных, состоящий из **λ-выражения** и **среды**, относительно которой выражение вычисляется, конкретней это:

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

Т.е, 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

# Замыкания в Python

```
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
```

# 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

# 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 << " ";
}
```

# Примеры “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() );
```

# 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