Объектно-ориентированное программирование

Object-oriented programming

I. Обзор курса

Overview

Почему объектно-ориентированное программирование

1. Дань моде
   a. с 1970-х написано огромное количество ОО-кода
   b. многие языки программирования написаны по принципу “OOP first”
   c. многие популярные идиомы пришли из ООП
2. Future-proofing
   a. большинство компаний не успевает за трендами и все еще зациклено на ООП
   b. большинство языков, на которых вам придется писать код, являются ОО[1]
3. ОО дизайн требует определенного подхода к архитектуре приложений

“It requires a different way of thinking about decomposition, and it produces software architectures that are largely outside the realm of the structured design culture.”

G. Booch

leon.brinzan@iis.utm.md

https://youtu.be/u3uuGrptpBs?si=0fn7CkxSrHyW3ROr&t=4355

Почему **C++**

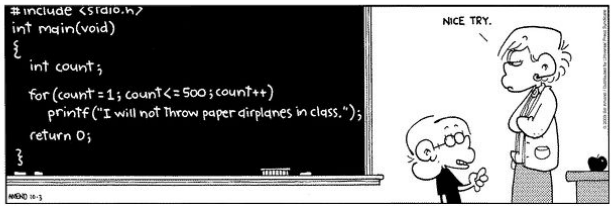
1. Потому что на **C** задания были бы слишком сложными
2. Потому что на **Python**/**Java**/**C#** задания были бы тривиальными
3. Потому что **Smalltalk**/**Erlang**/**Eiffel** недостаточно распространены
4. Потому что знать **C++** полезно (Linux, LLVM, Unity etc.)
5. Потому что цели этого курса не только в том, чтобы позволить понять ООП, а еще и в том, чтобы научить воспроизводить однажды понятое

Структура курса

Cod	Denumirea disciplinei/ modulului	Total	Contact direct	Studiu individual	Curs	Seminare	Lucrări practice	Lucrări de laborator
	Semestrul III							
F.O.010	Circuite și dispozitive electronice	150	75	75	45			30
D.O.003	Metode numerice	90	45	45	30			15
D.O.004	Programarea orientată pe obiecte	180	90	90	30		30	30

15 лекций – 33% посещаемости
7 лабораторных работ –
33% посещаемости
15% итоговой оценки
15 практических занятий –
33% посещаемости
15% итоговой оценки
2 промежуточные аттестации –
30% итоговой оценки
1 итоговый экзамен –
40% итоговой оценки

General

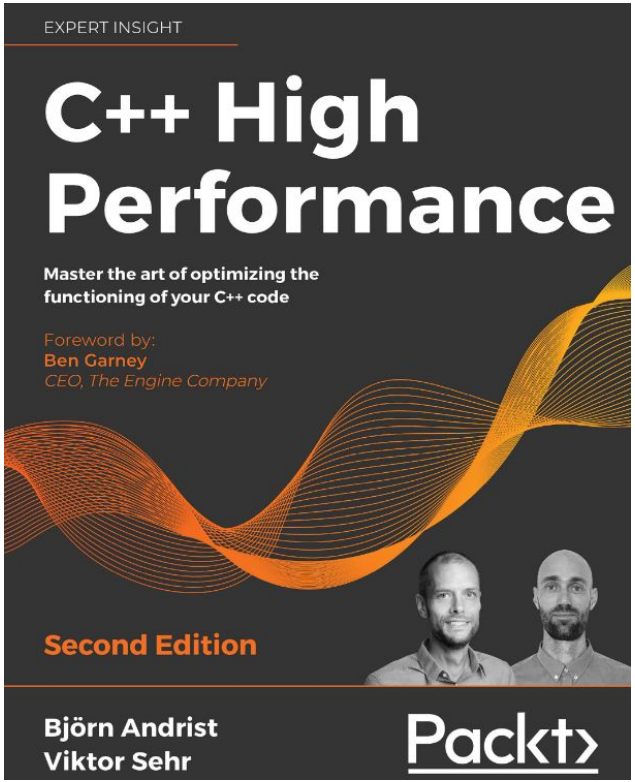


La finalizarea cu succes a acestei discipline, studenţii vor fi capabili să:
- utilizeze tehnicile de programare orientată pe obiecte (POO)
- implementeze în C++ tehnici ale POO
- utilizeze standard STL
- realizeze aplicaţii bazate pe POO.

Описание дисциплины 57[illegible]5KB документ PDF

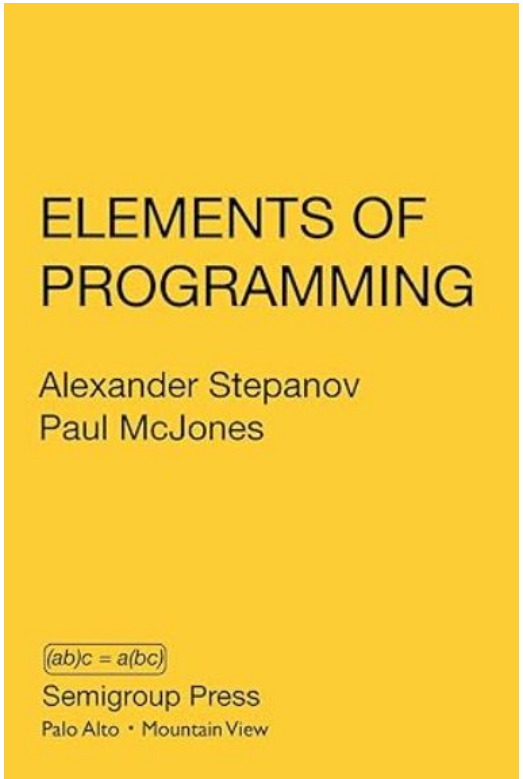
Методические указания к лабораторным работам 2.2MB документ PDF

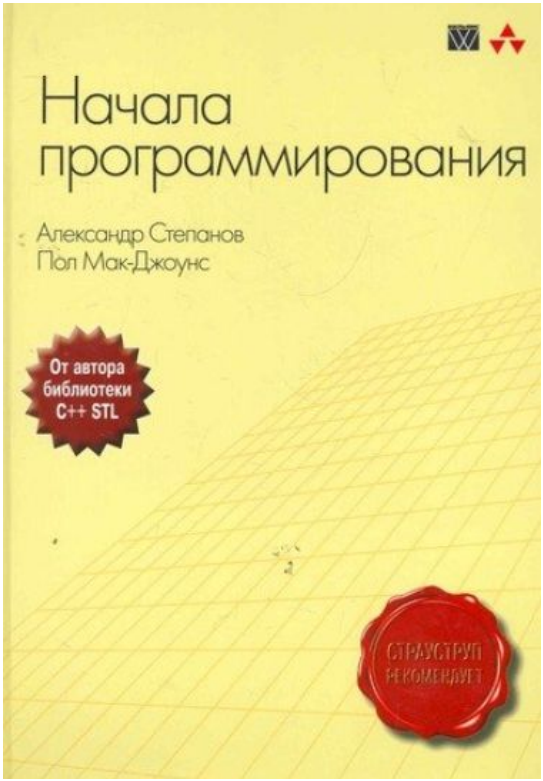
Что почитать



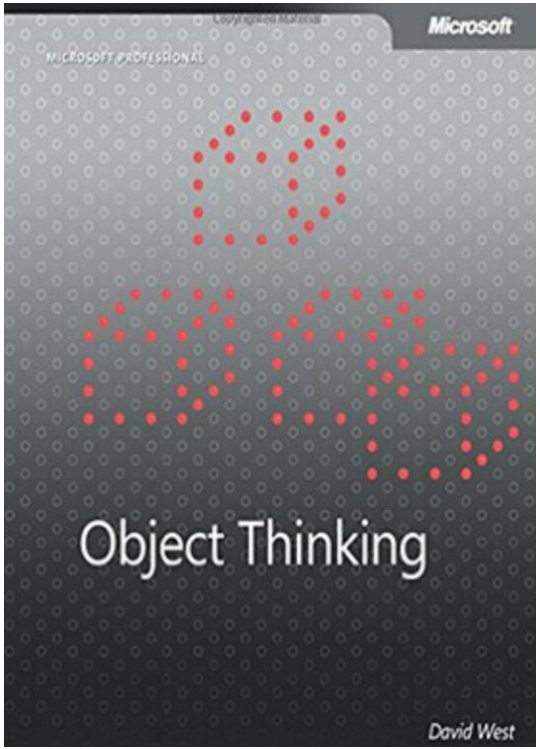
Что почитать

Дополнительные рекомендации

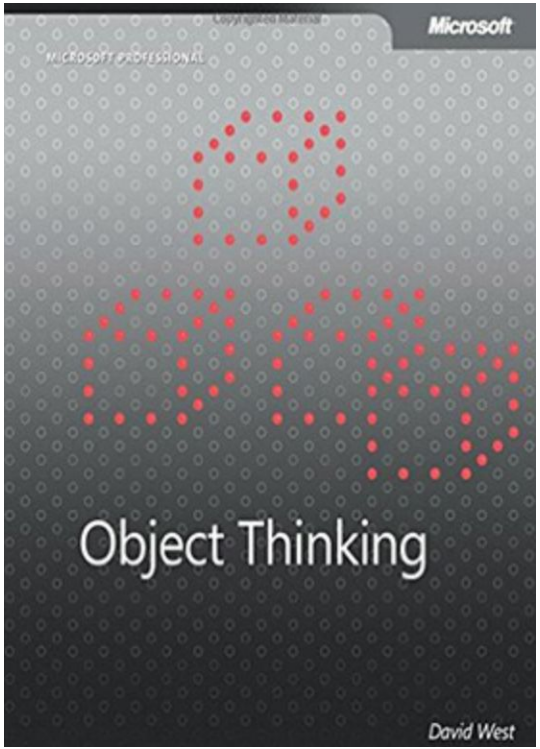




Дополнительные рекомендации



Дополнительные рекомендации



Yegor Bugayenko

★★★★★ **I'd like to give six stars to this book**

Reviewed in the United Kingdom on February 2, 2013

Verified Purchase

Great book about object oriented design, programming, analysis, and thinking. The book doesn't contain primers of Java code or any other practical examples. Instead, it gives a high-level overview of what is an object, how it differs from a data structure, how to think like an object, and why object thinking is a more effective approach than a procedural one. The book has a lot of references to other books, researches, ideas, and ideals. It is more a philosophical prophecy than a programmers guide.

rifraf

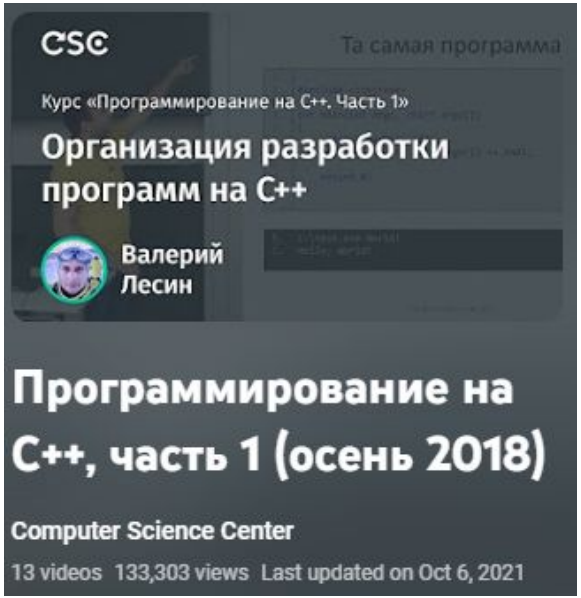
★☆☆☆☆ **I tried but...**

Reviewed in the United Kingdom on October 17, 2015

Verified Purchase

Recommended by Avdi Grimm, but sorry I didn't get it. Still no idea what Object Thinking is. Maybe I am spoiled by reading Uncle Bob - a great communicator.

Материалы для практических занятий



https://www.youtube.com/playlist?list=PLlb7e2G7aSpTFea2FYxp7mFfbZW-xavhL
https://www.youtube.com/playlist?list=PL4_hYwCyhAvZATDL6xxWCrHDqBF50RfKW
https://www.youtube.com/playlist?list=PL3BR09unfgciJ1_K_E914nohpiOiHnpsK

Вопросы?

The Pen Is Mightier Than the Keyboard

Pam A. Mueller[1] and Daniel M. Oppenheimer[2]
[1]Princeton University and [2]University of California, Los Angeles

Abstract
Taking notes on laptops rather than in longhand is increasingly common. Many researchers have suggested that laptop note taking is less effective than longhand note taking for learning. Prior studies have primarily focused on students' capacity for multitasking and distraction when using laptops. The present research suggests that even when laptops are used solely to take notes, they may still be impairing learning because their use results in shallower processing. In three studies, we found that students who took notes on laptops performed worse on conceptual questions than students who took notes longhand. We show that whereas taking more notes can be beneficial, laptop note takers' tendency to transcribe lectures verbatim rather than processing information and reframing it in their own words is detrimental to learning.

Keywords
academic achievement, cognitive processes, memory, educational psychology, open data, open materials

Received 5/11/13; Revision accepted 1/16/14

https://journals.sagepub.com/doi/abs/10.1177/0956797614524581

Laptop multitasking hinders classroom learning

for both users and nearby peers

Faria Sana [a], Tina Weston [b,c], Nicholas J. Cepeda [b,c,*]

[a] McMaster University, Department of Psychology, Neuroscience, & Behaviour, 1280 Main Street West, Hamilton, ON L8S 4K1, Canada
[b] York University, Department of Psychology, 4700 Keele Street, Toronto, ON M3J 1P3, Canada
[c] York University, LaMarsh Centre for Child and Youth Research, 4700 Keele Street, Toronto, ON M3J 1P3, Canada

ARTICLE INFO

Article history:
Received 11 September 2012
Received in revised form
5 October 2012
Accepted 12 October 2012

Keywords:
Laptops
Multitasking
Attentional control
Pedagogy

ABSTRACT

Laptops are commonplace in university classrooms. In light of cognitive psychology theory on costs associated with multitasking, we examined the effects of in-class laptop use on student learning in a simulated classroom. We found that participants who multitasked on a laptop during a lecture scored lower on a test compared to those who did not multitask, and participants who were in direct view of a multitasking peer scored lower on a test compared to those who were not. The results demonstrate that multitasking on a laptop poses a significant distraction to both users and fellow students and can be detrimental to comprehension of lecture content.

https://www.sciencedirect.com/science/article/pii/S0360131512002254

II. Историческая справка

Historical context

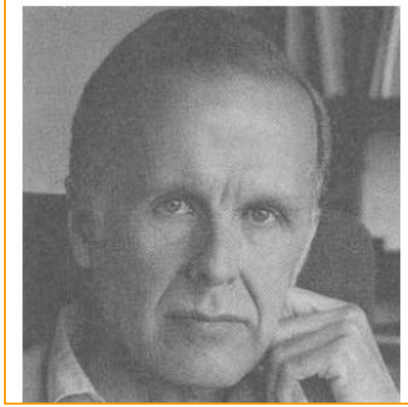
The core of what I now have to call "**real OOP**" – namely *encapsulated modules* *all the way down* *with pure messaging* – still hangs in there strongly because **it is nothing more than an abstract view of complex systems**.

Alan C. Kay

https://www.youtube.com/watch?v=zwucsB2EfXc?t=1029

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose



Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

http://worrydream.com/refs/Backus-CanProgrammingBeLiberated.pdf

Критерии к моделям исчисления

1. Наличие математического обоснования (описания).
2. Наличие системы хранения состояния программы (“истории”).
3. Семантический тип (state-transition, reduction).
4. Ясность и полезность (выражают ли программы процесс исчисления ясно; воплощают ли программы концепции, которые помогают формулировать процессы).

Классификация моделей исчисления

1. Простые (конечные автоматы, машины Тьюринга и т.п.).
2. Аппликативные (λ-исчисление, Lisp и т.п.).
3. Фон-Неймановские (компьютеры, Fortran и т.п.).
4. Другие (напр. data-flow-языки программирования).

Воронка фон Неймана (von Neumann bottleneck)

- Предназначение программы – изменить данные в хранилище
- Данные передаются из хранилища на ЦПУ только по шине
- Большая часть данных – мета-данные (адреса изменяемых значений)
- Как результат: программист находится в рамках мышления “**по одному слову за раз**” (переменные, прыжки между инструкциями, присваивание)

Скалярное произведение векторов

```
c := 0
for i := 1 step 1 until n do
    c := c + a[i] * b[i]
```

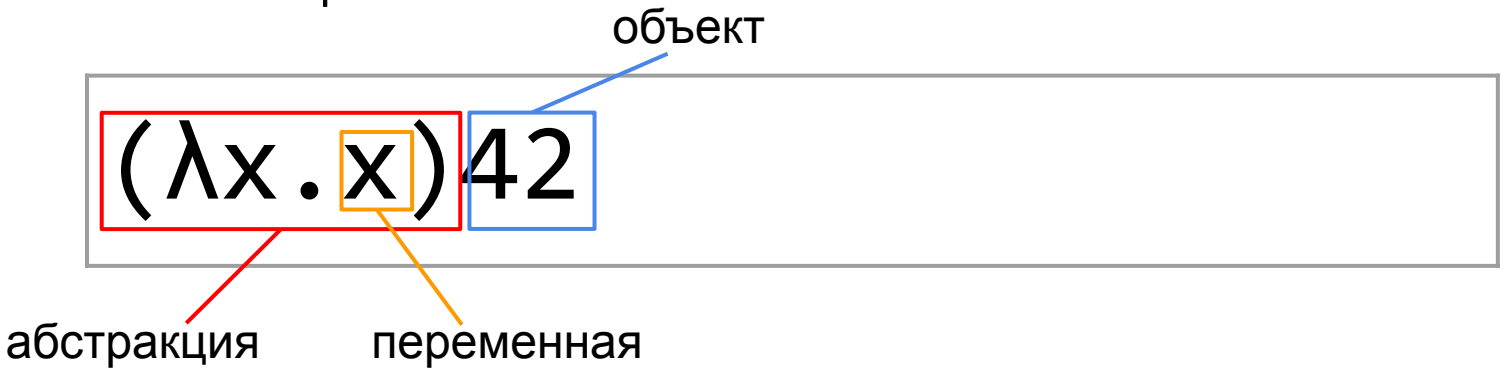
- Невидимое состояние
- Отсутствие иерархии (комбинации)
- Выражает повторение инструкции (трудно мысленно проследить)
- Состоит из повторных присваиваний результата выражения
- Работает только для векторов длины **n**

λ-выражения

1. Функциональная абстракция

λ-выражения

1. Функциональная абстракция
2. Аппликация



λ-выражения

Редукция

```
(λx.(λy.x)) 1 2
```

λ-выражения

β-редукция

```
(λx.(λy.x))(1)(2)
```

λ-выражения

β-редукция

```
(λx.(λy.x))(1)(2)
(λy.1)2
// 1
```

λ-выражения

Свободные (unbound) и зависимые (bound) переменные

```
λx.x          // x bound
λx.(λy.x)     // x bound
λx.(λy.z)     // z unbound
```

λ-выражения

Каким будет результат этих выражений?

```
id    <- λx.x
true  <- λx.(λy.x)
false <- λx.(λy.y)
((id) (false) (true))
```

λ-выражения

```
((id) (false) (true)) true
(true false true)
(λx.(λy.x)) false true
false
```

λ-выражения

https://www.youtube.com/watch?v=3VQ382QG-y4

Альтернативный стиль

1. Функциональный стиль **без использования переменных.**
2. **Алгебра** функциональных программ.
3. **Формализованная** система (позволяет комбинировать стиль и алгебру системы для построения более сложных систем).
4. **Аппликативная** система состояний и переходов между ними.

Скалярное произведение векторов

```
(Insert +) ◦ (ApplyToAll *) ◦ Transpose
```

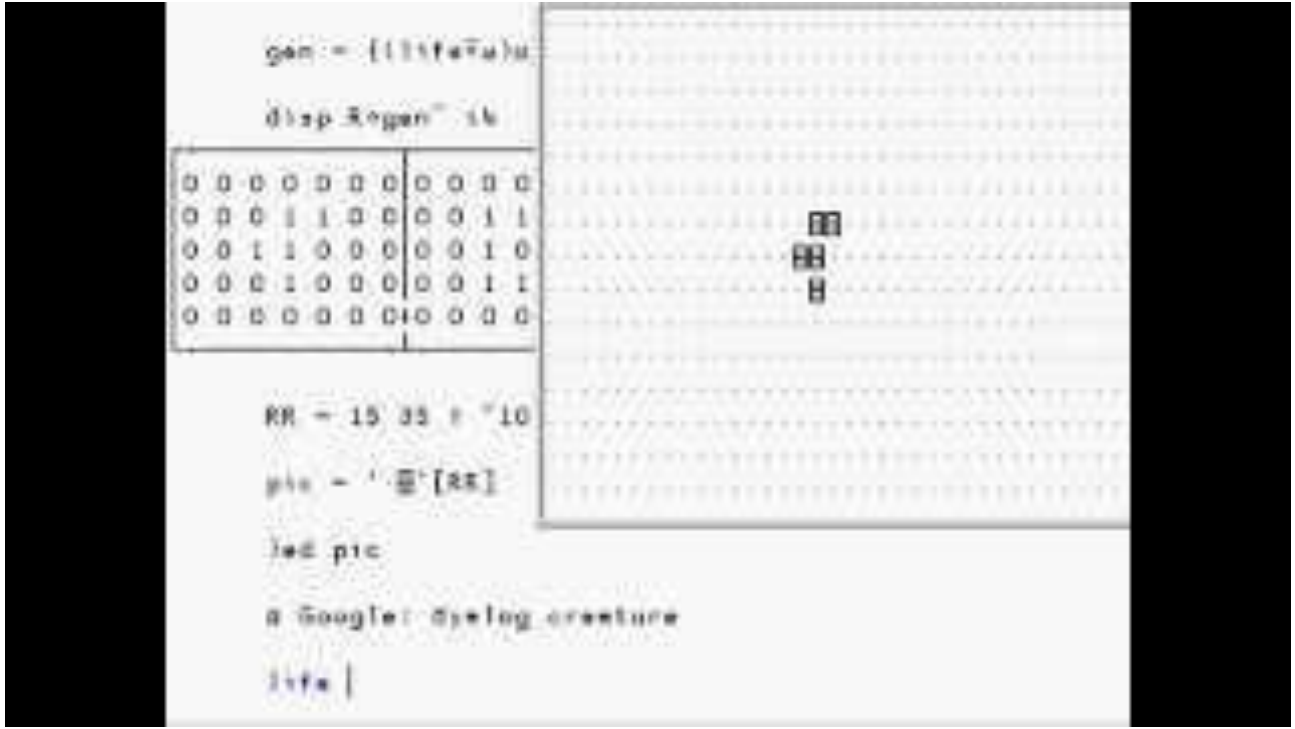
Например, для вектора {{1, 2, 3}, {6, 5, 4}}:

1. **Transpose** {{1, 2, 3}, {6, 5, 4}} : {{1, 6}, {2, 5}, {3, 4}}
2. (**ApplyToAll ***) {{1, 6}, {2, 5}, {3, 4}} : {6, 10, 12}
3. (**Insert +**) {6, 10, 12} : (**+** {6, (**+** {10, 12})})
4. **+** {6, 22} : 28

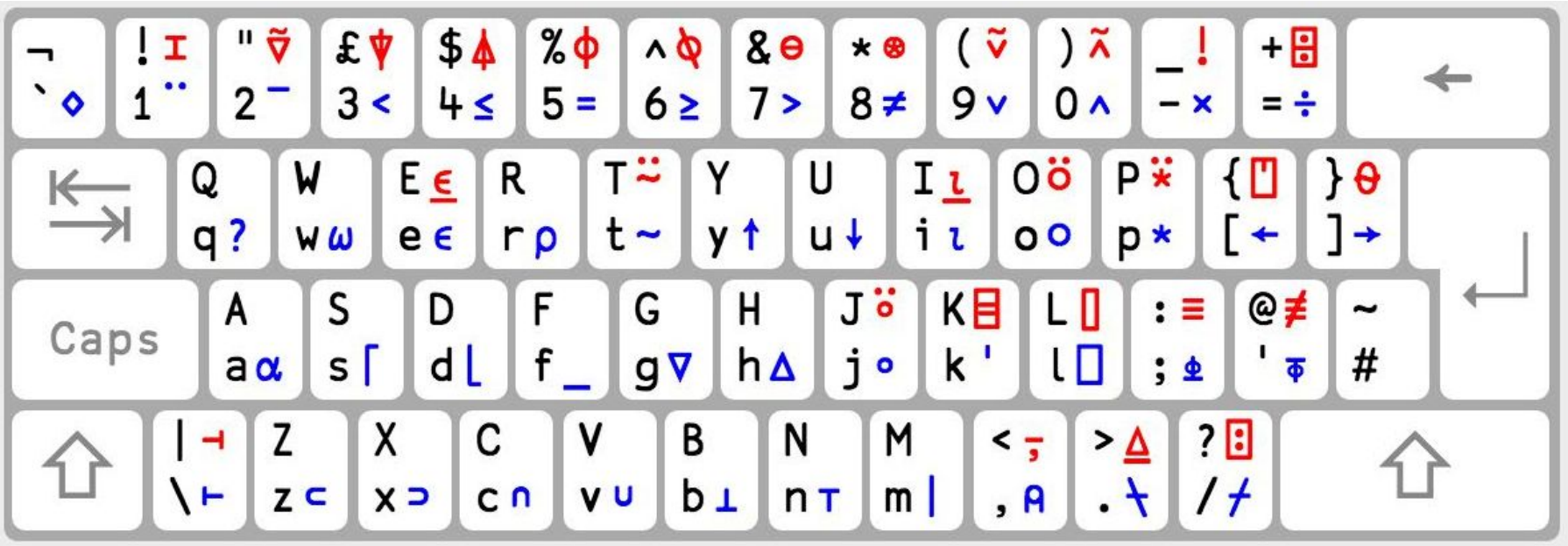
- Отсутствие состояния
- Есть строгая иерархия (композиция)
- Оперирует цельными концепциями, а не машинными словами
- Полностью обобщена (нет привязки к конкретным именам, длине векторов и т.д.)

APL (A Programming Language)

https://www.youtube.com/watch?v=a9xAKttWgP4



APL (A Programming Language)



https://www.softwarepreservation.org/projects/apl/Books/APROGRAMMING%20LANGUAGE

Lisp (**lis**t **p**rocessor)

"A Lisp programmer knows the value of everything, but the cost of nothing."

A. Perlis

- Любые частичные рекурсивные функции могут быть выражены в языке (Turing-complete)
- Использует λ-исчисление в качестве базовой модели
- Не имеет утверждений, только выражения (следовательно - нет побочных эффектов)
- Код и данные в программе взаимозаменяемы, так как представлены списками

https://www.cs.tufts.edu/~nr/cs257/archive/john-mccarthy/recursive.pdf

Lisp (**lis**t **p**rocessor)

Для выражения **λx.(x² + y)**:

```
(lambda (x) (+ (square x) y))
```

```
(define find (lambda (x y)
    (cond ((equal y nil) nil)
          ((equal x (car y)) x)
    (true (find x (cdr y)))
)))
```

Критерии к языкам программирования

```
i := 0
while i != f(i) do
    i := g(i)
```

- Вычислимость (computability, частичные ф-ции, “проблема остановки”)
- Статический анализ (compile-time/run-time)
- Производительность (efficiency, i.e. влияние управления памятью)
- Выразительность (expressiveness)

https://stanford-cs242.github.io/f19/

III. Введение в C++

Intro to C++

https://en.cppreference.com/w/

- **Language**
  - Keywords — Preprocessor
  - ASCII chart
  - Basic concepts
    - Comments
    - Names (lookup)
    - Types (fundamental types)
    - The main function
  - Expressions
    - Value categories
    - Evaluation order
    - Operators (precedence)
    - Conversions — Literals
  - Statements
    - if — switch
    - for — range-for (C++11)
    - while — do-while
  - Declarations — Initialization
  - Functions — Overloading
  - Classes (unions)
  - Templates — Exceptions
  - Freestanding implementations
- **Standard library (headers)**
- **Named requirements**
- **Feature test macros** (C++20)
- **Language support library**
  - Program utilities
  - source_location (C++20)
  - Coroutine support (C++20)
  - Three-way comparison (C++20)
  - Type support
  - numeric_limits — type_info
  - initializer_list (C++11)
- **Concepts library** (C++20)
- **Diagnostics library**
  - exception — System error
  - basic_stacktrace (C++23)
- **Memory management library**
  - unique_ptr (C++11)
  - shared_ptr (C++11)
  - weak_ptr (C++11)
  - Memory resources (C++17)
  - Allocators — Low level management
- **Metaprogramming library** (C++11)
  - Type traits — ratio
  - integer_sequence (C++14)
- **General utilities library**
  - Function objects — hash (C++11)
  - Swap — Type operations (C++11)
  - Integer comparison (C++20)
  - pair — tuple (C++11)
  - optional (C++17)
  - expected (C++23)
  - variant (C++17) — any (C++17)
  - String conversions (C++17)
  - Formatting (C++20)
  - bitset — Bit manipulation (C++20)
  - Debugging support (C++26)
- **Strings library**
  - basic_string — char_traits
  - basic_string_view (C++17)
  - Null-terminated strings:
    - byte — multibyte — wide
- **Containers library**
  - vector — deque — array (C++11)
  - list — forward_list (C++11)
  - map — multimap — set — multiset
  - unordered_map (C++11)
  - unordered_multimap (C++11)
  - unordered_set (C++11)
  - unordered_multiset (C++11)
  - Container adaptors
  - span (C++20) — mdspan (C++23)
- **Iterators library**
- **Ranges library** (C++20)
- **Algorithms library**
  - Execution policies (C++17)
  - Constrained algorithms (C++20)
- **Numerics library**
  - Common math functions
  - Mathematical special functions (C++17)
  - Mathematical constants (C++20)
  - Basic linear algebra algorithms (C++26)
  - Numeric algorithms
  - Pseudo-random number generation
  - Floating-point environment (C++11)
  - complex — valarray
- **Date and time library**
  - Calendar (C++20) — Time zone (C++20)
- **Localization library**
  - locale — Character classification
  - text_encoding (C++26)
- **Input/output library**
  - Print functions (C++23)
  - Stream-based I/O — I/O manipulators
  - basic_istream — basic_ostream
  - Synchronized output (C++20)
  - File systems (C++17)
- **Regular expressions library** (C++11)
  - basic_regex — Algorithms
  - Default regular expression grammar
- **Concurrency support library** (C++11)
  - thread — jthread (C++20)
  - atomic — atomic_flag
  - atomic_ref (C++20) — memory_order
  - Mutual exclusion — Semaphores (C++20)
  - Condition variables — Futures
  - latch (C++20) — barrier (C++20)
  - Safe Reclamation (C++26)

Основные отличия **C++** от **C**

1. `#include <cstdio>` или `#include <stdio.h>`
2. **`using namespace`** `std;`
3. **`class`** или **`struct`**
4. указатели или ссылки
5. **`new`** или **`malloc`** (**`delete`**, **`delete[]`** или **`free`**)
6. Конструктор, деструктор
7. `std::`**`array<int,`** `n>` или **`int`**`[n]` (или **VLA**)
8. `std::cout` или **`printf`**

leon.brinzan@iis.utm.md

#include <cstdio> или #include <stdio.h>

Оба варианта правильны, второй сохранился в языке для обратной совместимости, его не рекомендуется использовать из-за особенностей работы линкера **C++**

C compatibility headers

For some of the C standard library headers of the form *xxx*.h, the C++ standard library both includes an identically-named header and another header of the form c*xxx* (all meaningful c*xxx* headers are listed above). The intended use of headers of form *xxx*.h is for interoperability only. It is possible that C++ source files need to include one of these headers in order to be valid ISO C. Source files that are not intended to also be valid ISO C should not use any of the C headers.

With the exception of `complex.h`, each *xxx*.h header included in the C++ standard library places in the global namespace each name that the corresponding c*xxx* header would have placed in the `std` namespace.

These headers are allowed to also declare the same names in the `std` namespace, and the corresponding c*xxx* headers are allowed to also declare the same names in the global namespace: including `<cstdlib>` definitely provides `std::malloc` and may also provide `::malloc`. Including `<stdlib.h>` definitely provides `::malloc` and may also provide `std::malloc`. This applies even to functions and function overloads that are not part of C standard library.

https://en.cppreference.com/w/cpp/header

using namespace std;

Ключевое слово `namespace` служит для разрешения **конфликтов имен** в программе. При комбинировании нескольких файлов одно и то же слово, используемое в разных файлах будет внесено в общий файл дважды. Для решения этой проблемы в **C++** все имена объявляются в специальных блоках, которые называются *пространствами имен*

Notes

The using-directive `using namespace std;` at any namespace scope introduces every name from the namespace `std` into the global namespace (since the global namespace is the nearest namespace that contains both `std` and any user-declared namespace), which may lead to undesirable name collisions. This, and other using directives are generally considered bad practice at file scope of a header file (SF.7: Don't write `using namespace` at global scope in a header file).

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

using namespace std;

```
// x.h
#define A 10
```

```
// y.h
#define A 20
```

```
#include <cstdio>
#include "x.h"
#include "y.h"
int main() {
    printf("%d ", A);
}
```

```
// x.h
namespace x {
enum {A = 10};
}
```

```
// y.h
namespace y {
enum {A = 20};
}
```

```
#include <cstdio>
#include "x.h"
#include "y.h"
int main() {
    printf("%d ", A);
    printf("%d ", x::A);
}
```

`class` или `struct`

В языке **C**:

A struct is a type consisting of a sequence of members whose storage is allocated in an ordered sequence (as opposed to union, which is a type consisting of a sequence of members whose storage overlaps).

The type specifier for a struct is identical to the `union` type specifier except for the keyword used:

В языке **C++**:

Usage

- declaration of a compound type
- declaration of a scoped enumeration type (since C++11)
- If a function or a variable exists in scope with the name identical to the name of a non-union class type, `struct` can be prepended to the name for disambiguation, resulting in an elaborated type specifier.

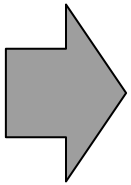
https://en.cppreference.com/w/cpp/keyword/struct

class или struct

```
class a {
    int A = 10;
};
#include <cstdio>
int main() {
    a x;
    printf("%d ", x.A);
}
```

class или struct

```
class a {
    int A = 10;
};
#include <cstdio>
int main() {
    a x;
    printf("%d ", x.A);
} //error:
'int a::A' is private
within this context
```

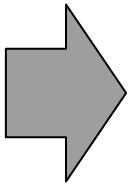


```
class a {
public:
    int A = 10;
};
#include <cstdio>
int main() {
    a x;
    printf("%d ", x.A);
}
```

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

class или struct

```
struct a {
    int A = 10;
};
#include <cstdio>
int main() {
    a x; // struct a
    printf("%d ", x.A);
}
```



```
struct a {
private:
    int A = 10;
};
#include <cstdio>
int main() {
    a x;
    printf("%d ", x.A);
} //error:
'int a::A' is private
within this context
```

Указатели или ссылки

Ссылка – это переменная, которая является псевдонимом уже **существующего** объекта или функции

1. Ссылки нельзя оставлять **не**инициализированными
2. Ссылки нельзя инициализировать **NULL**-значениями
3. Ссылки можно* инициализировать анонимными значениями

```
int i = 2;
int& ri = i;
int& rj = 2;
```

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

Указатели или ссылки

Ссылка – это переменная, которая является псевдонимом уже **существующего** объекта или функции

1. Ссылки нельзя оставлять **не**инициализированными
2. Ссылки нельзя инициализировать **NULL**-значениями
3. Ссылки можно* инициализировать анонимными значениями

```
int i = 2;
int& ri = i;
int& rj = 2; // error: cannot bind non-const lvalue
reference of type 'int&' to an rvalue of type 'int'
const int& rj = 2;
```

Указатели или ссылки

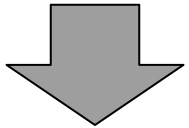
Ссылка – это переменная, которая является псевдонимом уже **существующего** объекта или функции

1. Ссылки нельзя оставлять **не**инициализированными
2. Ссылки нельзя инициализировать **NULL**-значениями
3. Ссылки можно* инициализировать анонимными значениями

```
int& a[3]; // error: declaration of 'a' as array of references
int&* p;   // error: cannot declare pointer to 'int&'
int& &r;   // error: 'r' declared as reference but not initialized
```

Категории значений (l-values/r-values)

```
void f(int&);
void g(int&&);
void h(const int&);
```



```
int i = 0;
f(i);
g(i); // an rvalue reference cannot be bound to an lvalue
h(i);
```

```
f(42); // initial value of reference to non-const must be an lvalue
g(42);
h(42);
```

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

new, new[], delete, delete[]

1. `new` – это ключевое слово
2. `new` – это также **функция**
3. `new` **запрашивает память** у ядра операционной системы аналогично `malloc`
4. `new` может вызвать **исключение**
5. `new` вызывает **конструктор** объекта

The new-expression allocates storage by calling the appropriate allocation function. If *type* is a non-array type, the name of the function is `operator new`. If *type* is an array type, the name of the function is `operator new[]`.

As described in allocation function, the C++ program may provide global and class-specific replacements for these functions. If the new-expression begins with the optional `::` operator, as in `::new T` or `::new T[n]`, class-specific replacements will be ignored (the function is looked up in global scope). Otherwise, if T is a class type, lookup begins in the class scope of T.

When calling the allocation function, the new-expression passes the number of bytes requested as the first argument, of type `std::size_t`, which is exactly `sizeof(T)` for non-array T.

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

`new`, `new[]`, `delete`, `delete[]`

1. `delete` – это ключевое слово
2. `delete` – это также **функция**
3. `delete` **возвращает память** ядру операционной системы аналогично `free`
4. `delete` **не** вызывает исключение в случае неправильного обращения, поведение не определено (**undefined behavior**, **UB**)
5. `delete` вызывает **деструктор** объекта

If `ptr` is not a null pointer and the deallocation function is not a destroying delete(since C++20), the delete-expression invokes the destructor (if any) for the object that is being destroyed, or for every element of the array being destroyed (proceeding from the last element to the first element of the array). The destructor must be accessible from the point where the delete-expression appears.

After that, whether or not an exception was thrown by any destructor, the delete-expression invokes the deallocation function: either `operator delete` (first version) or `operator delete[]` (second version), unless the matching new-expression was combined with another new-expression(since C++14).

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

new, new[], delete, delete[]

```
auto s = new int {42};
// int *s = malloc(sizeof(int));
auto a = new int [2] {0, 1};
// int *a = malloc(sizeof(int) * 2);
delete s;
// free(s);
delete [] a;
// free(a);
```

Конструктор, деструктор

Constructors are non-static member functions declared with a special declarator syntax, they are used to initialize objects of their class types.

In the definition of a constructor of a class, *member initializer list* specifies the initializers for direct and virtual bases and non-static data members. (Not to be confused with `std::initializer_list`.)

```
class a {
    int A = 10;
public:
    a() { }
    a(int x) { this->A = x; }
    a(double x) : A(x) { }
};
```

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

Конструктор, деструктор

The destructor is called whenever an object's lifetime ends, which includes

- program termination, for objects with static storage duration
- thread exit, for objects with thread-local storage duration (since C++11)
- end of scope, for objects with automatic storage duration and for temporaries whose life was extended by binding to a reference
- delete-expression, for objects with dynamic storage duration
- end of the full expression, for nameless temporaries
- stack unwinding, for objects with automatic storage duration when an exception escapes their block, uncaught.

```
class a {
    int *p {nullptr};
public:
    ~a() { delete[] p; }
    a(size_t n) { this->p = new int[n]; }
};
```

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

std::array<int, n>

`std::array` is a container that encapsulates fixed size arrays.

This container is an aggregate type with the same semantics as a struct holding a C-style array `T[N]` as its only non-static data member. Unlike a C-style array, it doesn't decay to `T*` automatically. As an aggregate type, it can be initialized with aggregate-initialization given at most N initializers that are convertible to T: `std::array<int, 3> a = {1, 2, 3};`.

```
std::array<int, 4> a = {1, 2, 3, 4};
// int a[4] = {1, 2, 3, 4};
a[0] = 0;
```

https://en.cppreference.com/w/cpp/container/array

std::array<int, n>

```
template <typename T, size_t n> class array {
    T p[n]{0};
public:
    ~array() = default;
    array(const std::initializer_list<T>& l) {
        std::copy(l.begin(), l.end(), p);
    }
};

std::array<int, 4> a = {1, 2, 3, 4};
// int a[4] = {1, 2, 3, 4};
a[0] = 0;
```

- Procedural: *I can write functions then call them*
- OOP: *I can call functions that haven't been written yet*
- Templates: *I can design with types that haven't been written yet*

std::cout или printf

1. std::cout[1] – это **не** функция

```
/// iosfwd: Base class for @c char output streams.
typedef basic_ostream<char> ostream;
#include <ostream>
namespace std _GLIBCXX_VISIBILITY(default) { // iostream:
    extern istream cin;  /// Linked to standard input
    extern ostream cout; /// Linked to standard output
    extern ostream cerr; /// Linked to standard error (unbuffered)
    extern ostream clog; /// Linked to standard error (buffered)
} // namespace
```

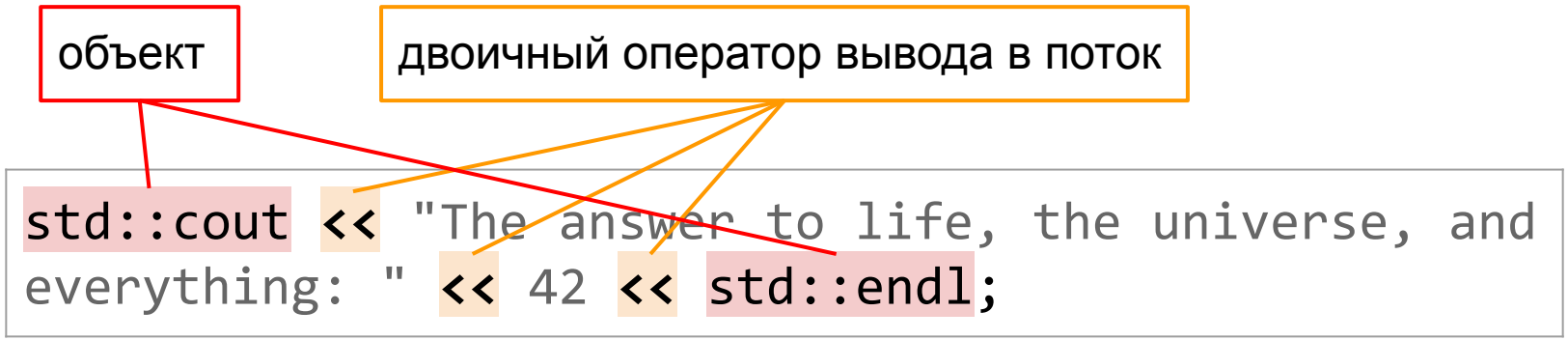
2. std::cout имеет доступ к списку имен функций и операторов, которые можно вызвать для вывода
3. функции вывода указаны в классе basic_ostream как свойства

std::cout или printf

```
#include <cstdio>
printf("The answer to life, the universe, and
everything: %d\n", 42);

#include <iostream>
std::cout << "The answer to life, the universe, and
everything: " << 42 << std::endl;
```

std::cout или printf



std::cout или printf

```
std::cout.operator<<("The answer to life, the
universe, and everything:
").operator<<(42).operator<<(std::endl);
```


```
std::cout << "The answer to life, the universe, and
everything: " << 42 << std::endl;
```


leon.brinzan@iis.utm.md

Object-oriented programming