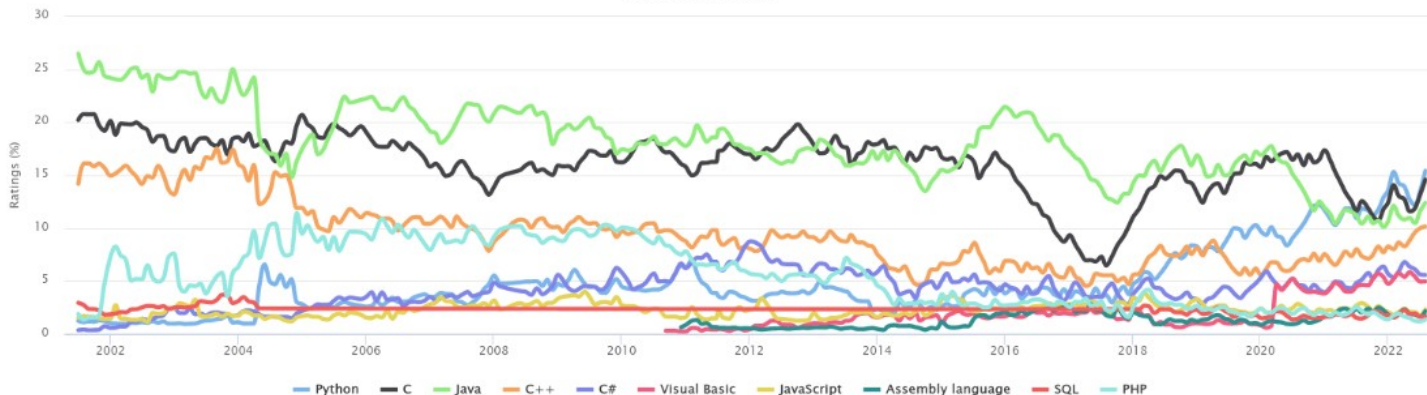


Способы задания однозначности в архитектурах ВС. Связь с типизацией

TIOBE Programming Community Index

Source: www.tiobe.com



Факторы, влияющие на каждый из архитектурных уровней

1. Методы алгоритмизации (МА)

- императивное программирование
- функциональное программирование
- автоматное программирование
- декларативное программирование

2. Методы композиции (МК)

- абстрактные типы данных + функции
- классы = данные + методы
- модули
- пространства имен

3. Методы задания однозначности (МЗО)

- статическая типизация (однозначность на уровне описания типов данных)
- динамическая типизация (однозначность на уровне вычисляемых тегов)
- операционная однозначность (на уровне кодов операций компьютера)

4. Методы управления вычислениями (МУВ)

- последовательное программирование
- параллельное программирование (разнообразии вариантов)

5. Уровни абстракции (УА)

- Непосредственное отображение
- Абстракция типов
- Метапрограммирование

...

Начальный взгляд

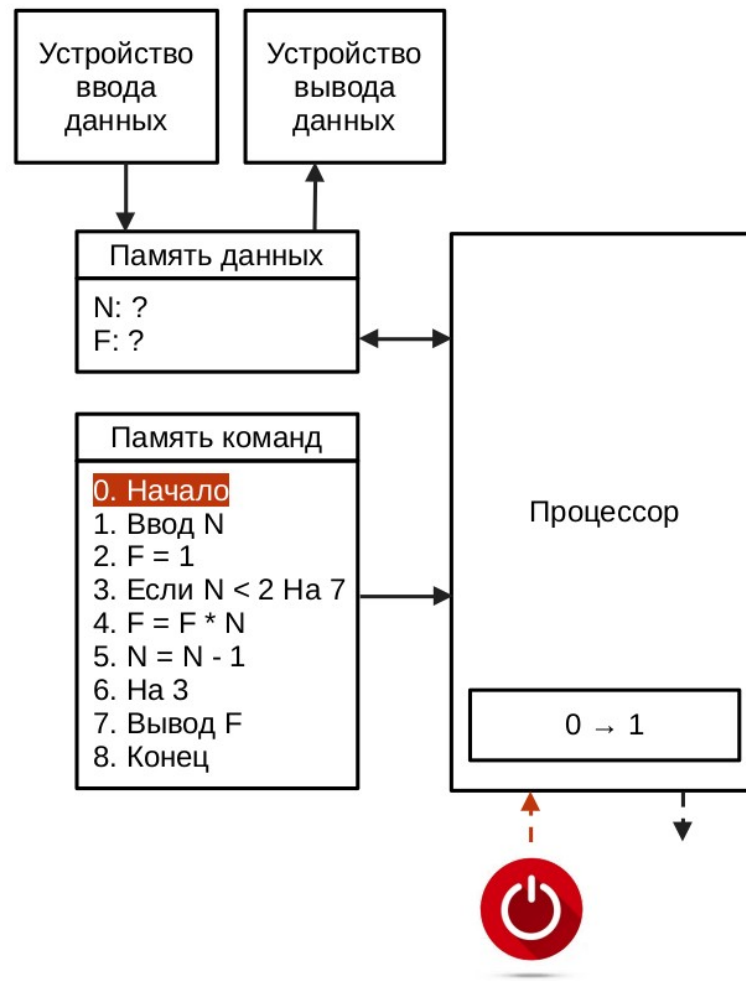
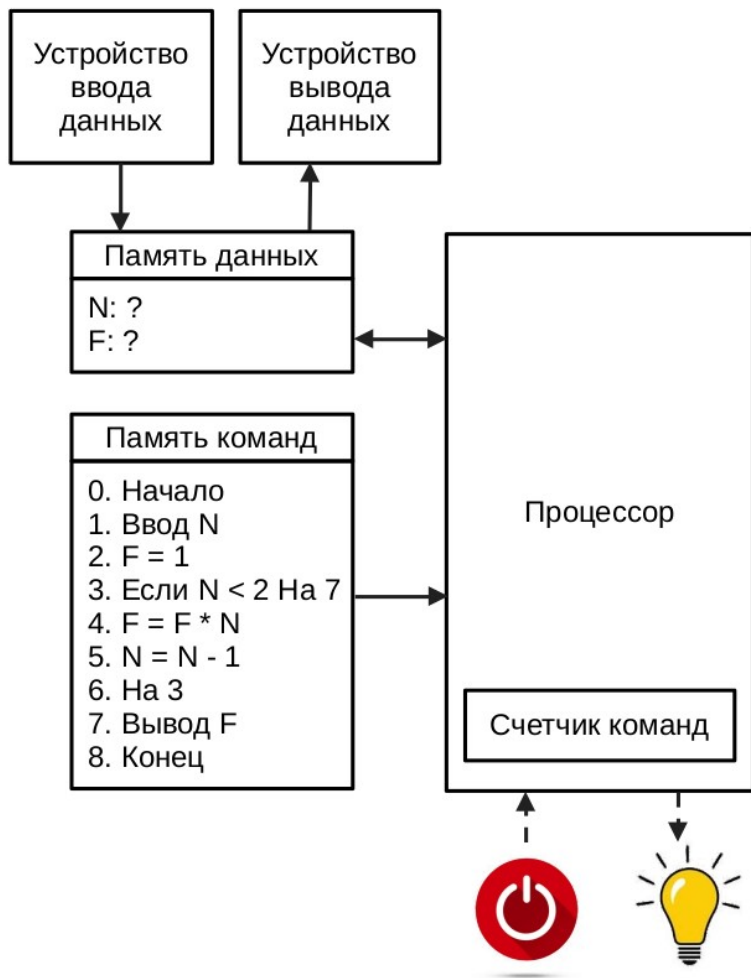
1. Начало
2. Конец
3. Ввод
4. Вывод
5. Операция
6. Условный переход
7. Безусловный переход



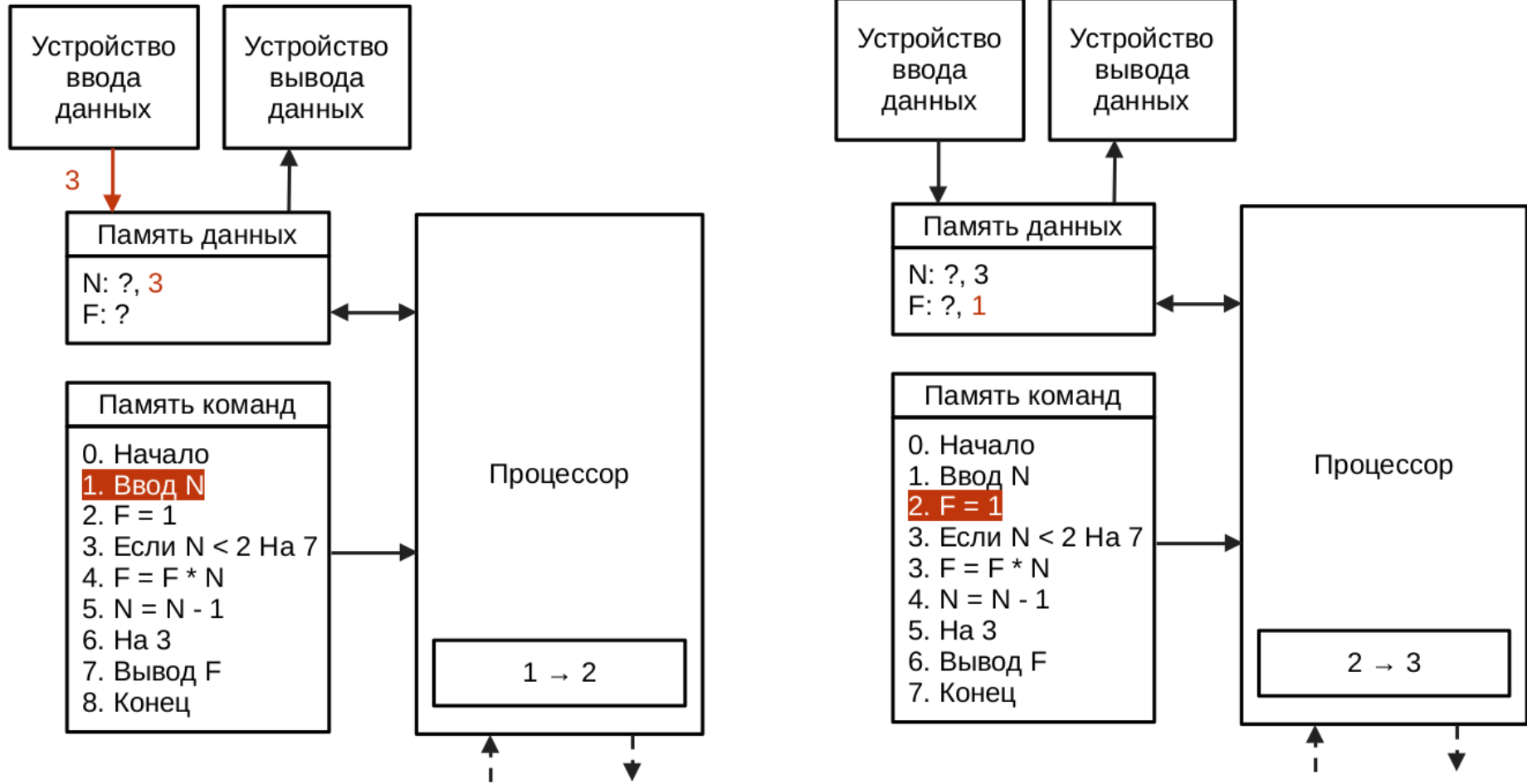
Что внутри?

0. Начало
1. Ввод N
2. $F = 1$
3. Если $N < 2$ На 7
4. $F = F * N$
5. $N = N - 1$
6. На 3
7. Вывод F
8. Конец

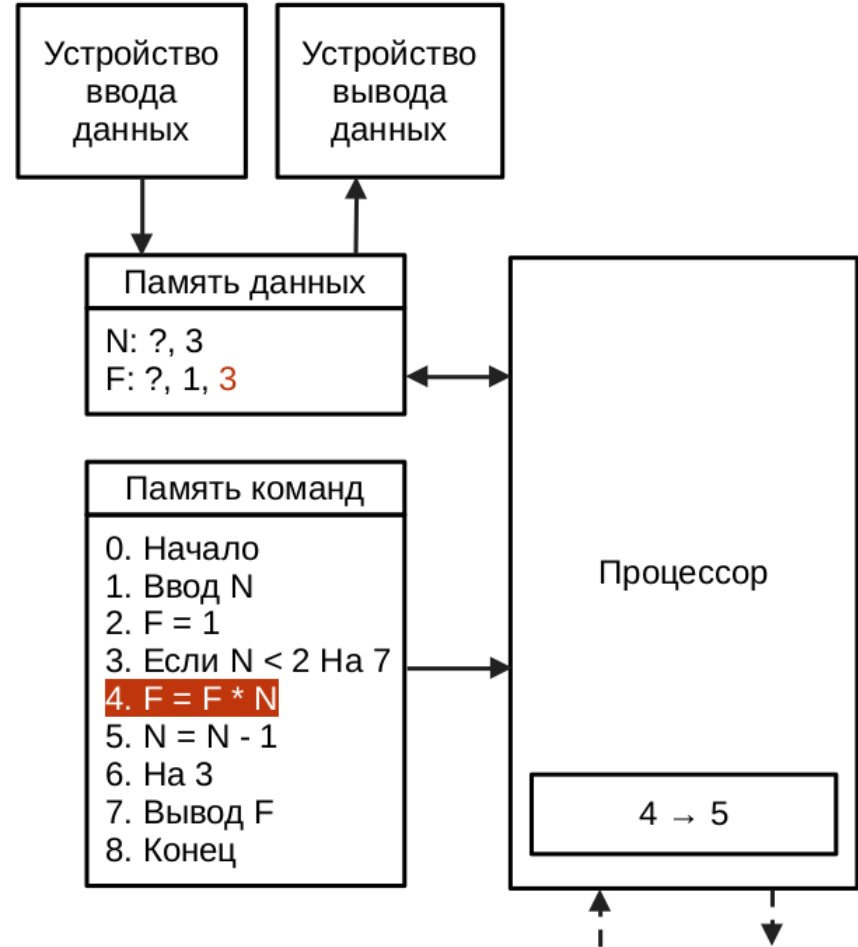
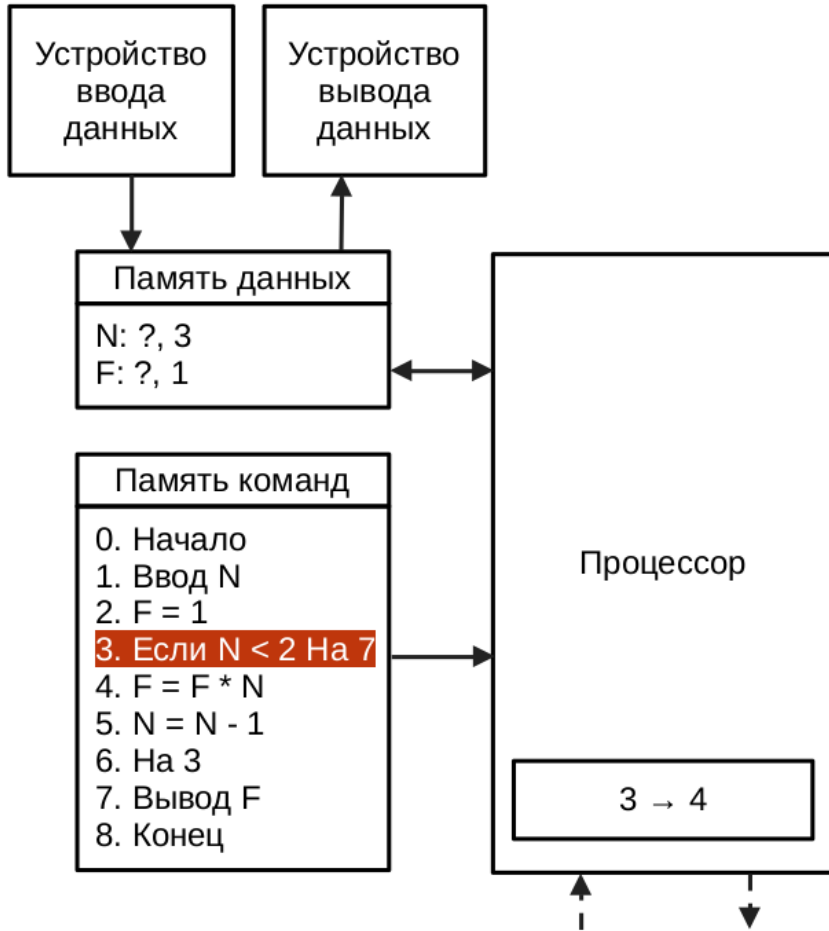
Что внутри?



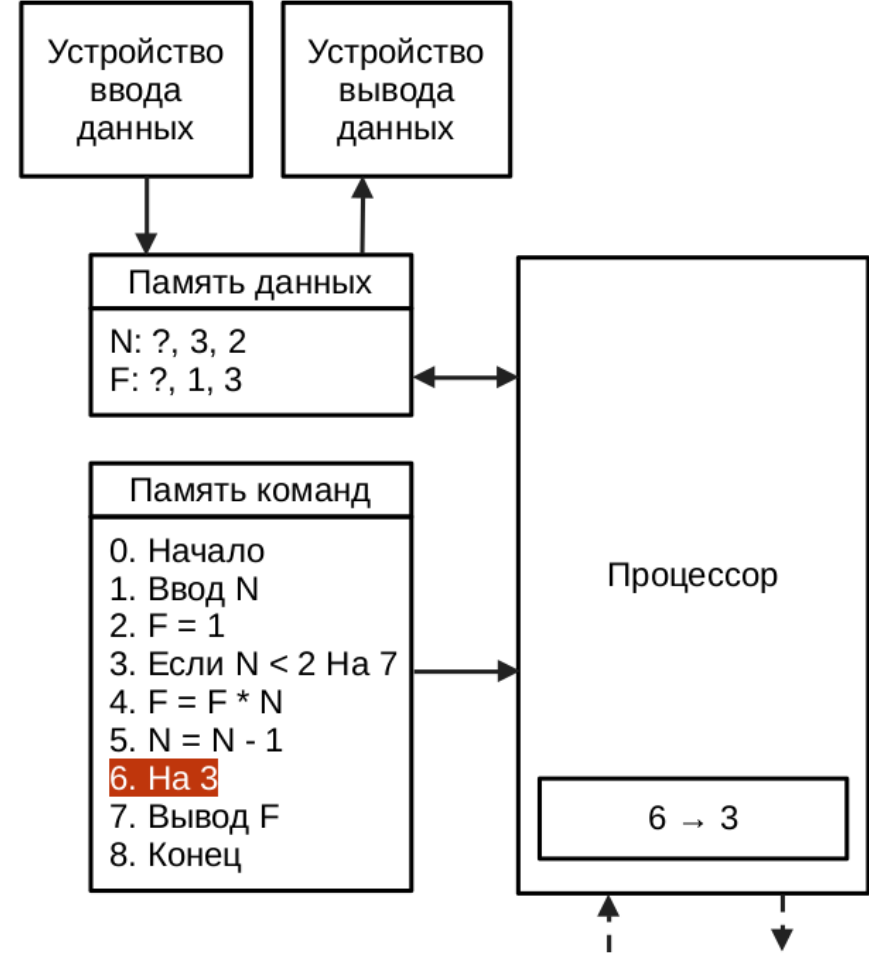
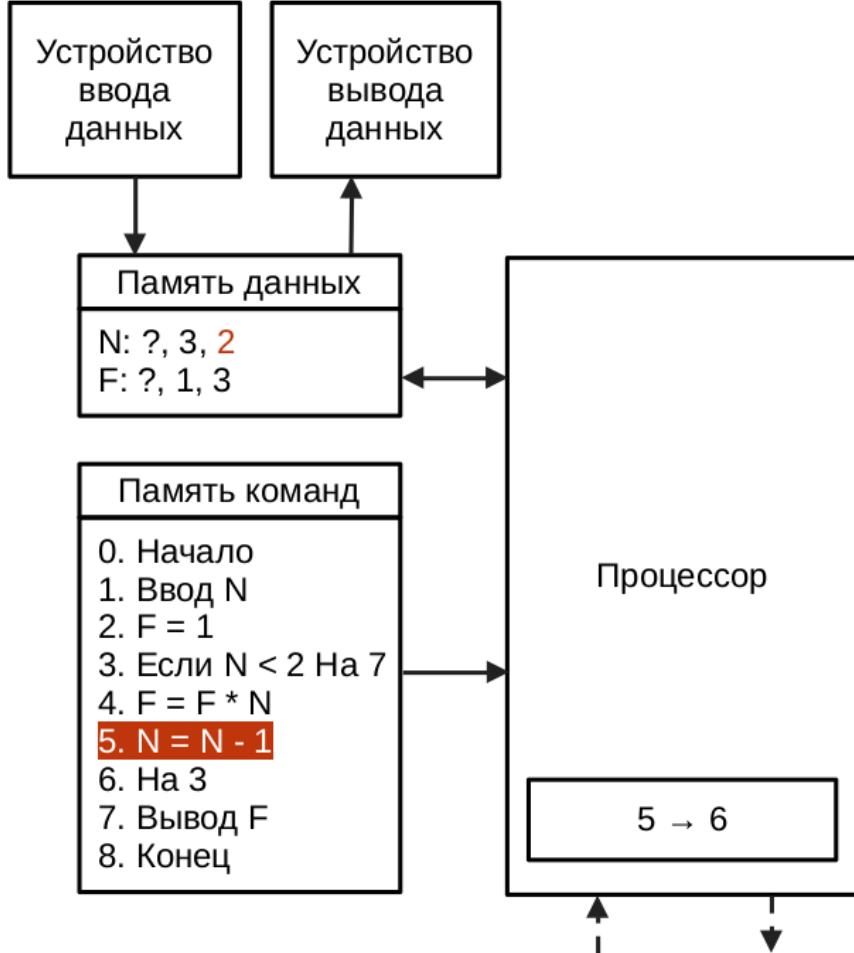
Что внутри?



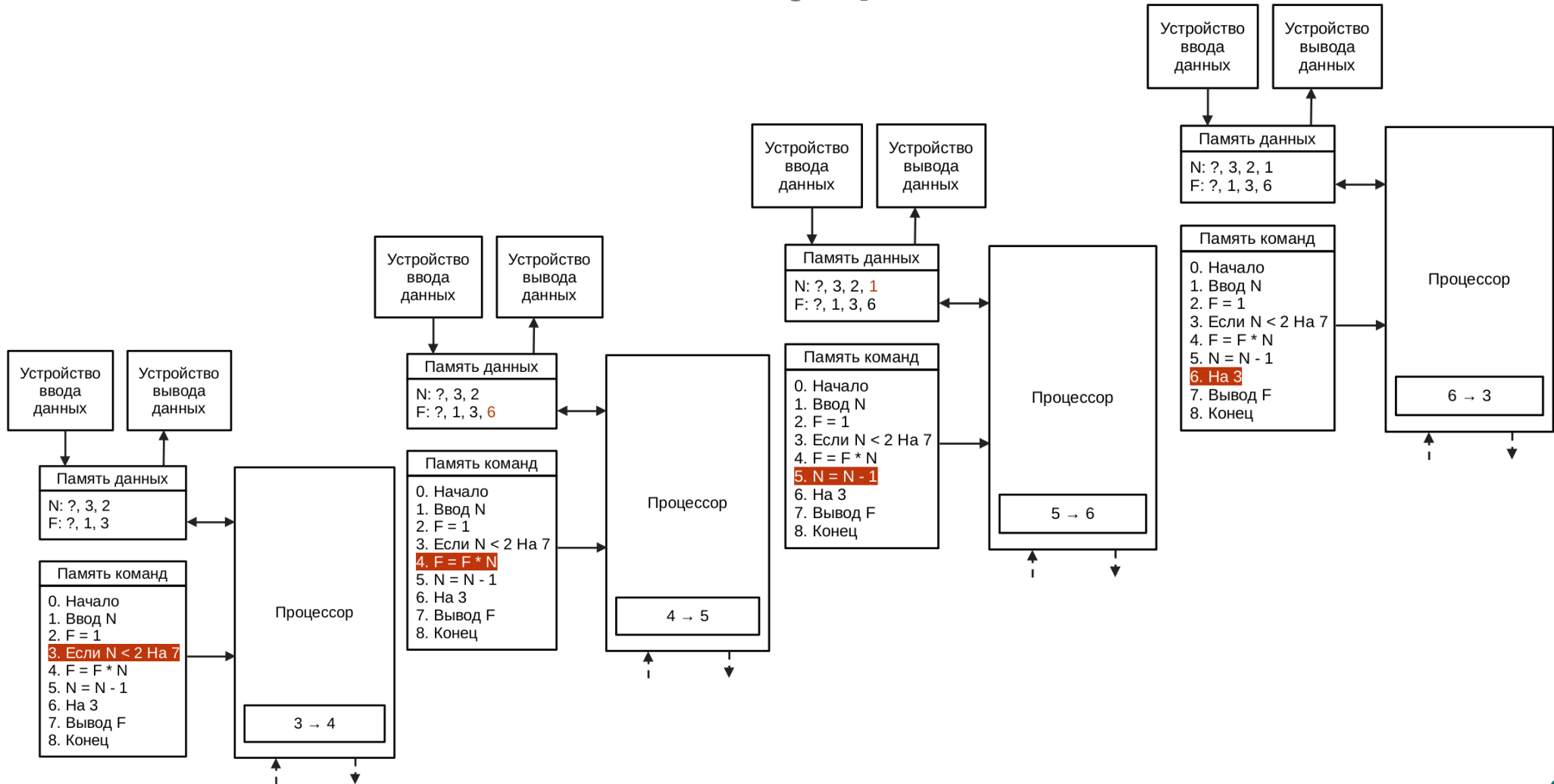
Что внутри?



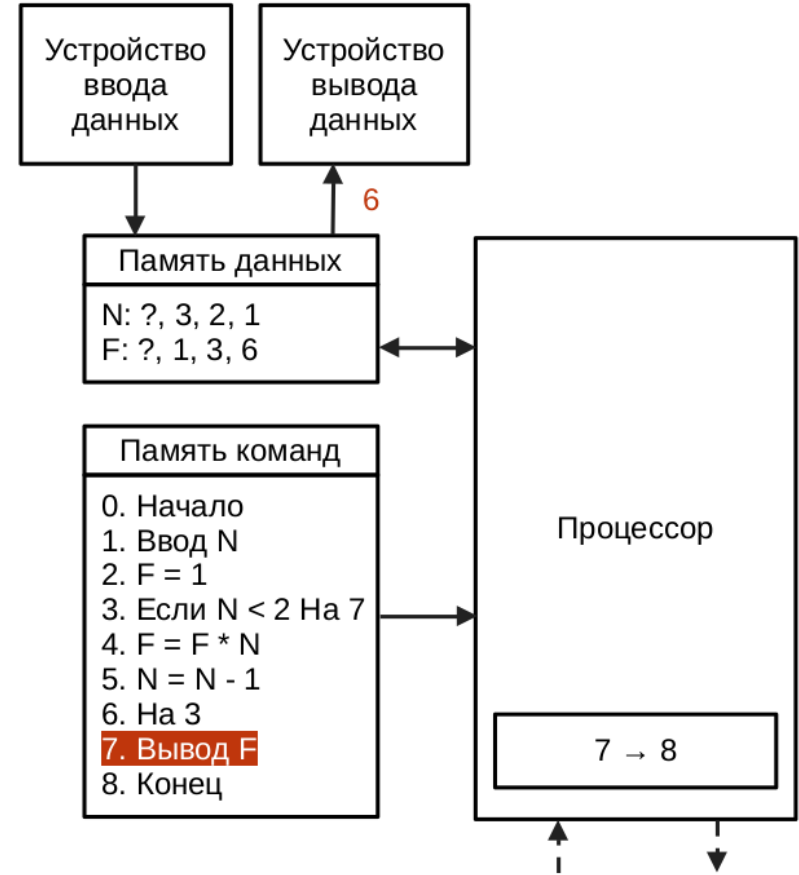
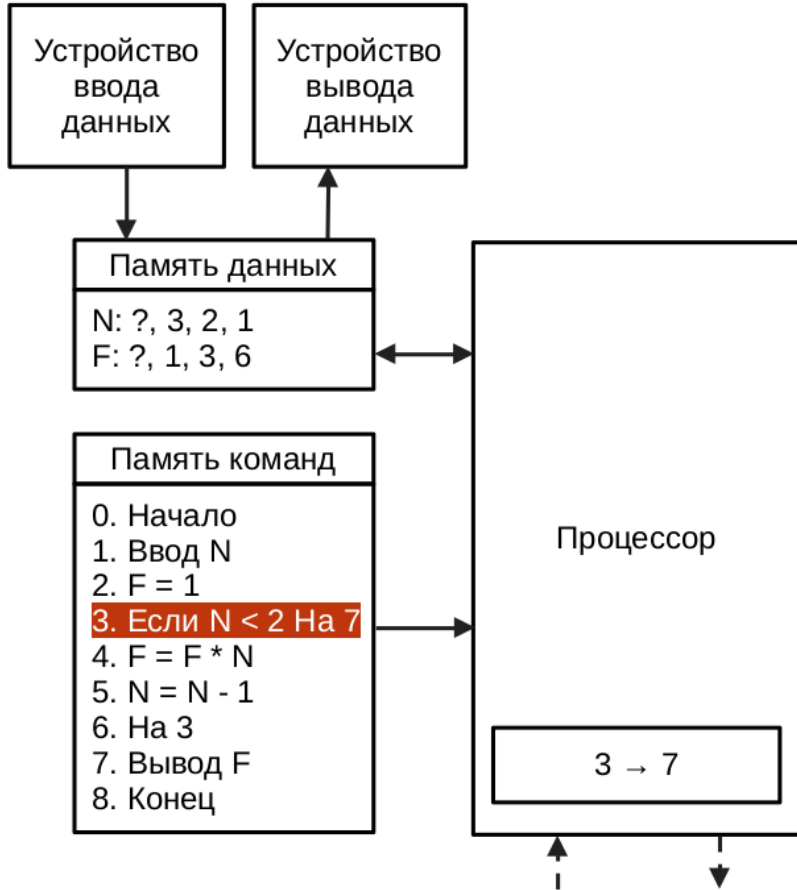
Что внутри?



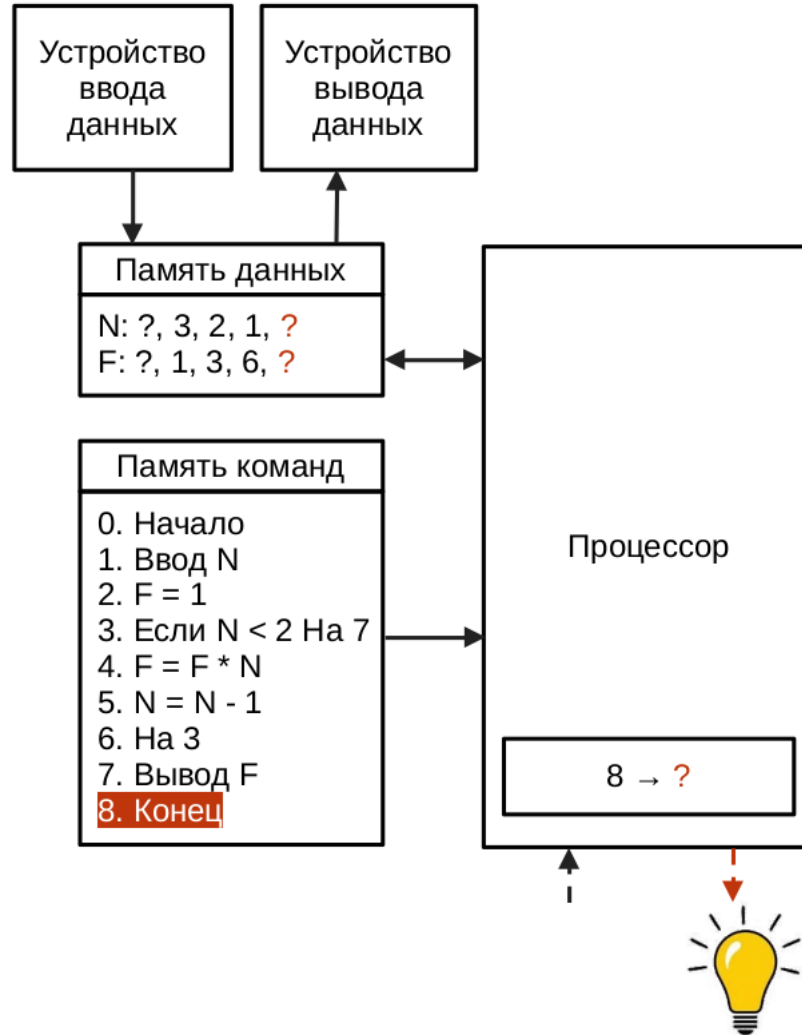
Что внутри?



Что внутри?



Что внутри?



А компьютер выполнит этот алгоритм?

- 1) Можно ли использовать действительные числа, символы, строки?
- 2) Как воспринимается ввод данных?
- 3) Как отображаются данные?
- 4) На какие устройства ввода-вывода можно использовать?
- 5) Какова семантика каждой выполняемой операции?
- 6) Какой тип памяти данных у N и F?

Способы задания однозначности

Однозначность определяет четкие правила выполнения операций реальными и виртуальными вычислительными системами, позволяя избегать или обходить ошибки программирования.

Различные методы задания однозначности операций позволяют **контролировать корректность программы** с разной степенью и на различных стадиях обработки

Выделяются:

- 1) **Операционная однозначность** (бестиповые системы)
- 2) **Динамическая однозначность** (системы с динамической типизацией)
- 3) **Статическая однозначность** (системы со статической типизацией)

Операционная однозначность

Однозначность операций формируется за счет четкого определения что и с какими типами данных делает каждая операция. Сами данные при этом не несут никакой дополнительной семантической идентификации и представляются в виде набора строк бит (байт), размещенных в памяти. Доступ к обезличенным данным осуществляется по адресам, задаваемым в операциях. Для таких архитектур характерны бестиповые языки.

Примеры подобных архитектур:

- 1) Современные архитектуры уровня системы команд и их языки ассемблера
- 2) Объектно-ориентированный язык программирования Eolang
- 3) Языки системного программирования

Программа для компьютера

```
1  #include <stdio.h>
2
3  static int n;
4  static int f = 1;
5
6  int main() {
7      printf("n? ");
8      scanf("%d", &n);
9      loop:
10     | if(n < 2) goto end;
11         f *= n;
12         n--;
13         goto loop;
14     end:
15     | printf("n! = %d\n", f);
16         return 0;
17     }
```

Бестиповое программирование на GNU Assembler (AT&T)

```
1 # asm-fact.s
2 # Константные данные
3 .section .rodata
4 question:
5 .string "n? "
6 .equ questionLength, .-question-1
7 formatIn:
8 .string "%d"
9 .equ formatInLength, .-formatIn-1
10 formatOut:
11 .string "n! = %d\n"
12 .equ formatOutLength, .-formatOut-1
13
14 # Статические переменные
15 .data
16 n: .long 0
17
18 # Текст программы
19 .text
20 .globl main
21 main:
22 pushq %rbp # пролог
23 movq %rsp, %rbp
24
25 # Ввод начального значения n
26 leaq question(%rip), %rdi # адрес формата подсказки
27 movl $0, %eax # не действительные числа
28 call printf@PLT # печать подсказки
29
```

```
30 leaq formatIn(%rip), %rdi # адрес формата числа
31 leaq n(%rip), %rsi # не действительные числа
32 movl $0, %eax # ввод целого
33 call scanf@PLT
34
35 # Вычисление факториала
36 movl $1, %eax # начальная установка f
37 movl n(%rip), %ebx # перенос n в регистр
38 loop:
39 cmpl $2, %ebx # проверка на завершение
40 jl end # выход по меньше
41 mull %ebx # f *= n
42 decl %ebx # --n;
43 jmp loop
44 end:
45
46 # Вывод результата вычислений
47 leaq formatOut(%rip), %rdi # адрес формата результата
48 movq %rax, %rsi # значение результата
49 movl $0, %eax # не действительные числа
50 call printf@PLT # печать результата
51
52 movl $0, %eax # return 0
53 popq %rbp # эпилог
54 ret
```


Бестиповое программирование на GNU Assembler (Intel)

```
1 # asm-fact.s
2     .intel_syntax noprefix
3 # Константные данные
4     .section .rodata
5 question:
6     .string "n? "
7     .equ   questionLength, .-question-1
8 formatIn:
9     .string "%d"
10    .equ   formatInLength, .-formatIn-1
11 formatOut:
12    .string "n! = %d\n"
13    .equ   formatOutLength, .-formatOut-1
14
15 # Статические переменные
16     .data
17 n:     .long 0
18
19 # Текст программы
20     .text
21     .globl main
22 main:
23     push    rbp                # пролог
24     mov     rbp, rsp
25
26     # Ввод начального значения n
27     lea    rdi, question[rip]  # адрес формата подсказки
28     mov     eax, 0             # не действительные числа
29
30
31     call   printf@plt          # печать подсказки
32
33     lea    rdi, formatIn[rip]  # адрес формата числа
34     lea    rsi, n[rip]
35     mov     eax, 0             # не действительные числа
36     call   scanf@plt          # ввод целого
37
38     # Вычисление факториала
39     mov     eax, 1             # начальная установка f
40     mov     ebx, n[rip]        # перенос n в регистр
41 loop:
42     cmp     ebx, 2             # проверка на завершение
43     jl     end                 # выход по меньше
44     mul    ebx                 # f *= n
45     dec    ebx                 # --n;
46     jmp    loop
47
48     # Вывод результата вычислений
49     lea    rdi, formatOut[rip] # адрес формата результата
50     mov     rsi, rax            # значение результата
51     mov     eax, 0             # не действительные числа
52     call   printf@plt          # печать результата
53
54     mov     eax, 0             # return 0
55     pop     rbp                # эпилог
56     ret
```

Динамическая однозначность

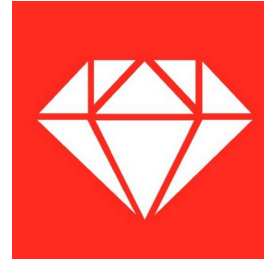
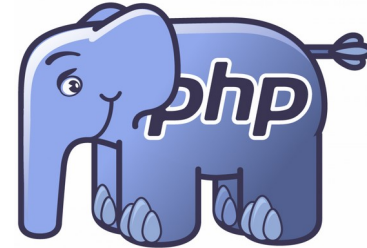
Динамическая однозначность операций формируется за счет того, что с каждым значением, формируемым в программе сопоставляется его тип. Любая операция над данным может проверить этот тип и выбрать в соответствии с этим нужные вычисления. То есть, одна и та же операция может обрабатывать различные типы данных. При этом идентификация типа осуществляется во время выполнения программы. Одни и те же переменные могут хранить данные различного типа. В любой момент программа может проверить тип переменной. Данный подход широко используется в языках программирования, ориентированных на интерпретацию.

Динамическая типизация — приём, используемый в языках программирования и языках спецификации, при котором переменная связывается с типом в момент присваивания значения, а не в момент объявления переменной. Таким образом, в различных участках программы одна и та же переменная может принимать значения разных типов.

Свойства языков с динамической типизацией





















Примеры языков с динамической типизацией:

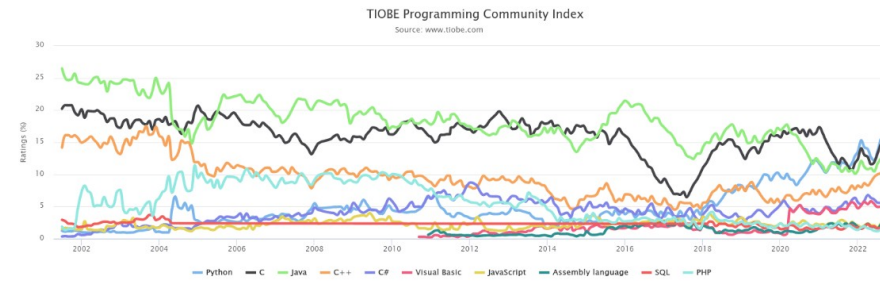
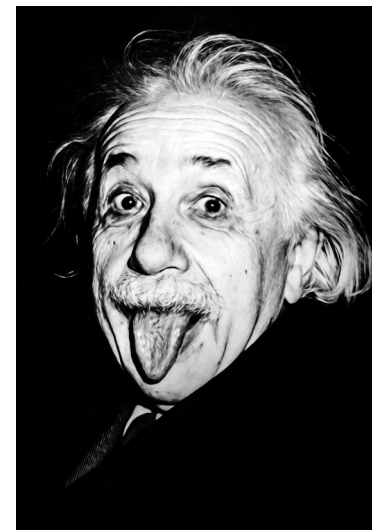
- Smalltalk
- Python
- Objective-C
- Ruby
- PHP
- Perl
- JavaScript
- Лисп



Динамическая типизация упрощает написание программ для работы с меняющимся окружением, при работе с данными переменных типов; при этом отсутствие информации о типе на этапе компиляции повышает вероятность ошибок в исполняемых модулях.

Позиции языков с динамической типизацией

Aug 2022	Aug 2021	Change	Programming Language	Ratings	Change
1	2	▲	 Python	15.42%	+3.56%
2	1	▼	 C	14.59%	+2.03%
3	3		 Java	12.40%	+1.96%
4	4		 C++	10.17%	+2.81%
5	5		 C#	5.59%	+0.45%
6	6		 Visual Basic	4.99%	+0.33%
7	7		 JavaScript	2.33%	-0.61%
8	9	▲	 Assembly language	2.17%	+0.14%
9	10	▲	 SQL	1.70%	+0.23%
10	8	▼	 PHP	1.39%	-0.80%
11	16	▲▲	 Swift	1.27%	+0.30%
12	12		 Classic Visual Basic	1.27%	+0.04%
13	22	▲▲	 Delphi/Object Pascal	1.22%	+0.60%
14	23	▲	 Objective-C	1.22%	+0.61%
15	18	▲	 Go	0.98%	+0.08%
16	14	▼	 R	0.92%	-0.13%
17	17		 MATLAB	0.90%	-0.08%
18	15	▼	 Ruby	0.82%	-0.18%
19	13	▼▼	 Fortran	0.81%	-0.32%
20	20		 Perl	0.72%	-0.06%



Организация значения при динамической типизации

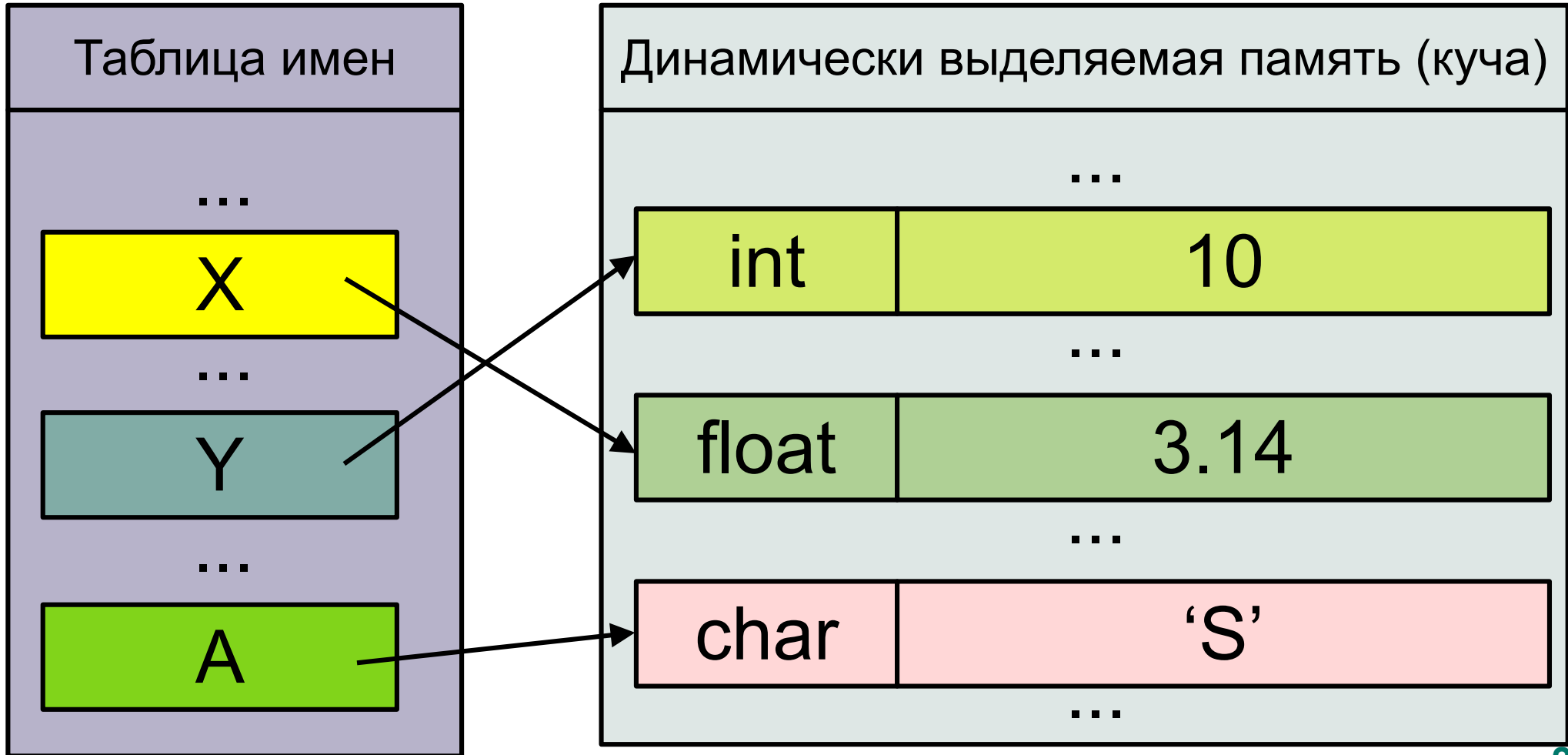
Тип	Величина
-----	----------

int	10
-----	----

float	3.14
-------	------

char	'S'
------	-----

Обращение к величинам через указатели



Python. Использование REPL (Read-Execute-Print Loop) для демонстрации изменения типа переменной

```
>>> value = 10
>>> value
10
>>> type(value)
<class 'int'>
>>> value = 3.14
>>> value
3.14
>>> type(value)
<class 'float'>
>>> value = "Hello!"
>>> value
'Hello!'
>>> type(value)
<class 'str'>
```


Python. Изменение и проверка типа в программе

```
1 import random
2
3 ▼ for i in range(10):
4     key = random.randint(1,2)
5 ▼     if key == 1:
6         value = random.uniform(1.0, 10.0)
7 ▼     else:
8         value = random.randint(100, 200)
9
10 print('key = {0}; value = {1}; type = {2}'.format(key, value, type(value)))
```

```
key = 2; value = 155; type = <class 'int'>
key = 2; value = 130; type = <class 'int'>
key = 1; value = 6.131331242406195; type = <class 'float'>
key = 1; value = 8.280520967840578; type = <class 'float'>
key = 1; value = 5.030964057739875; type = <class 'float'>
key = 2; value = 134; type = <class 'int'>
key = 1; value = 6.393939330816693; type = <class 'float'>
key = 2; value = 101; type = <class 'int'>
key = 1; value = 7.203902995902304; type = <class 'float'>
key = 2; value = 155; type = <class 'int'>
```

Python. Использование динамической однозначности

```
1 n = int(input("n? "))
2 f = 1
3 while n > 1:
4     f *= n
5     n -= 1
6 print("n! = {0}".format(f))
```

```
n? 5
n! = 120
```

```
1 n = float(input("n? "))
2 f = 1
3 while n > 1:
4     f *= n
5     n -= 1
6 print("n! = {0}".format(f))
```

```
n? 5
n! = 120.0
```

```
n? 4.8
n! = 91.929599999999998
```

Архитектура и статическая типизация



Статическая однозначность

Статическая однозначность операций формируется за счет того, что с каждым значением в программе сопоставляется его тип. Этот тип задается при описании переменных и может быть проверен во время компиляции. Для всех временных и промежуточных значений тип может быть также выведен во время компиляции. Поэтому его не имеет смысла проверять во время выполнения. Одна и та же операция может быть задана с разными типами, но все вопросы по ее конкретному выполнению решаются во время компиляции (статический полиморфизм). С каждой переменной сопоставляется только один тип. Допускает эффективную трансформацию в бестиповые архитектуры уровня системы команд. Используется в языках компилируемого типа.

Примеры:

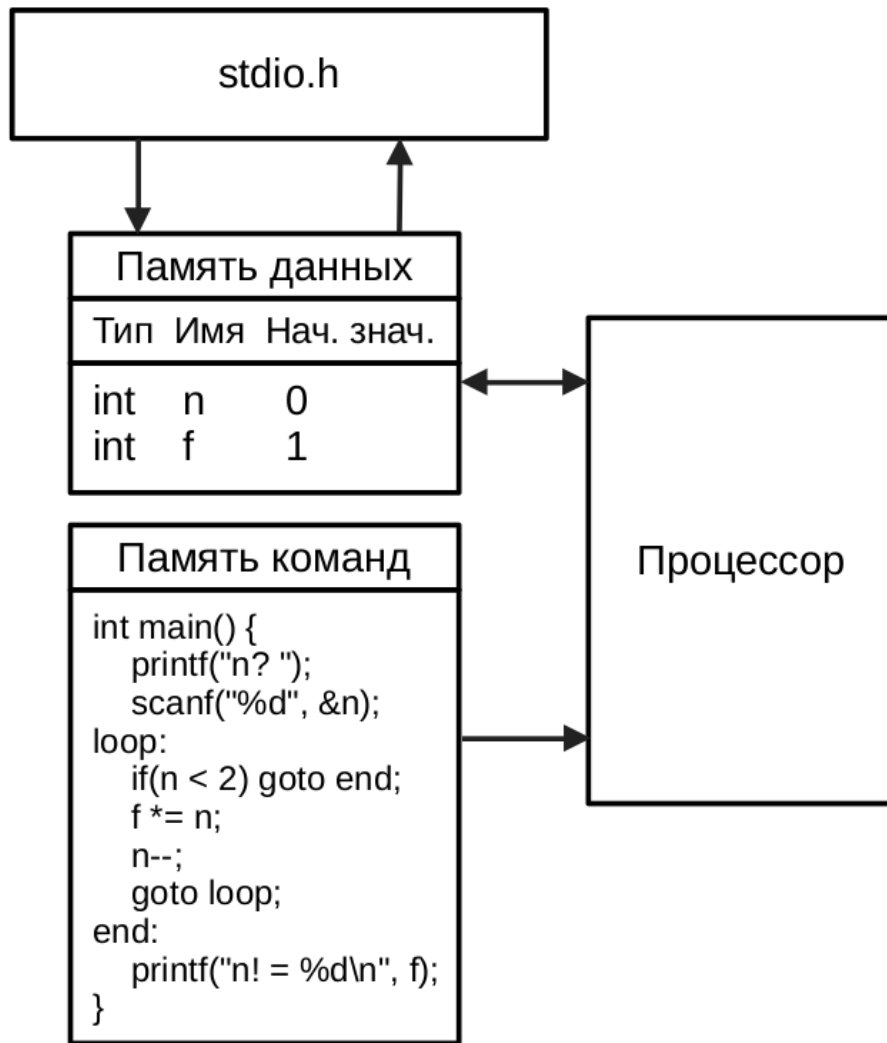
1) Императивные языки программирования: C, C++, Pascal, Oberon family, Java, C#, Rust, Go...

2) Языки функционального программирования: ML, Haskell...

Программа для компьютера

```
1  #include <stdio.h>
2
3  static int n;
4  static int f = 1;
5
6  int main() {
7      printf("n? ");
8      scanf("%d", &n);
9      loop:
10     | if(n < 2) goto end;
11         f *= n;
12         n--;
13         goto loop;
14     end:
15     | printf("n! = %d\n", f);
16         return 0;
17     }
```

Отображение программы на структуру



Формирование однозначности операций

```
1 #include <stdio.h>
2
3 static int n;
4 static int f = 1;
5
6 int main() {
7     // Во время выполнения:
8     printf("n? "); // calc(char*)
9     scanf("%d", &n); // calc(char*); if(%d)-> use n as int
10 loop:
11     // Во время компиляции:
12     if(n < 2) goto end; // <(int, int) -> bool
13     f *= n; // *(int, int) -> int; =(int) -> int
14     n--; // --(int) -> int
15     goto loop;
16 end:
17     // Во время выполнения:
18     printf("n! = %d\n", f); // calc(char*); if(%d)-> use n as int
19     return 0;
20 }
```

```
[ fact01]$ c++ fact.cpp
[ fact01]$ ./a.out
n? 5
n! = 120
```

Формирование однозначности операций

```
1 #include <stdio.h>
2
3 static int n;
4 static int f = 1;
5
6 int main() {
7     // Во время выполнения:
8     printf("n? "); // calc(char*)
9     scanf("%c", &n); // calc(char*); if(%c)-> use n as char
10 loop:
11     // Во время компиляции:
12     if(n < 2) goto end; // <(int, int) -> bool
13     f *= n; // *(int, int) -> int; =(int) -> int
14     n--; // --(int) -> int
15     goto loop;
16 end:
17     // Во время выполнения:
18     printf("n! = %s\n", f); // calc(char*); if(%s)-> use n as char*
19     return 0;
20 }
```

```
[fact01]$ c++ bad_fact.cpp
[fact01]$ ./a.out
n? 5
n! = (null)
```


Избыточность статической типизации

```
1 #include <stdio>
2
3 int n;
4 int f = 1;
5
6 int main() {
7     printf("n? ");
8     scanf("%d", &n);
9     loop:
10    if(n < 2) goto end;
11    f *= n;
12    n--;
13    goto loop;
14 end:
15    printf("n! = %d\n", f);
16 }
```

man 3 printf

man 3 scanf

int printf(char *, ...) →

int scanf(char *, ...) →

< (int, int) → bool

*(int, int) → int; = (int) → int

--(int) → int

**Трансформация к операционной
однозначности:**

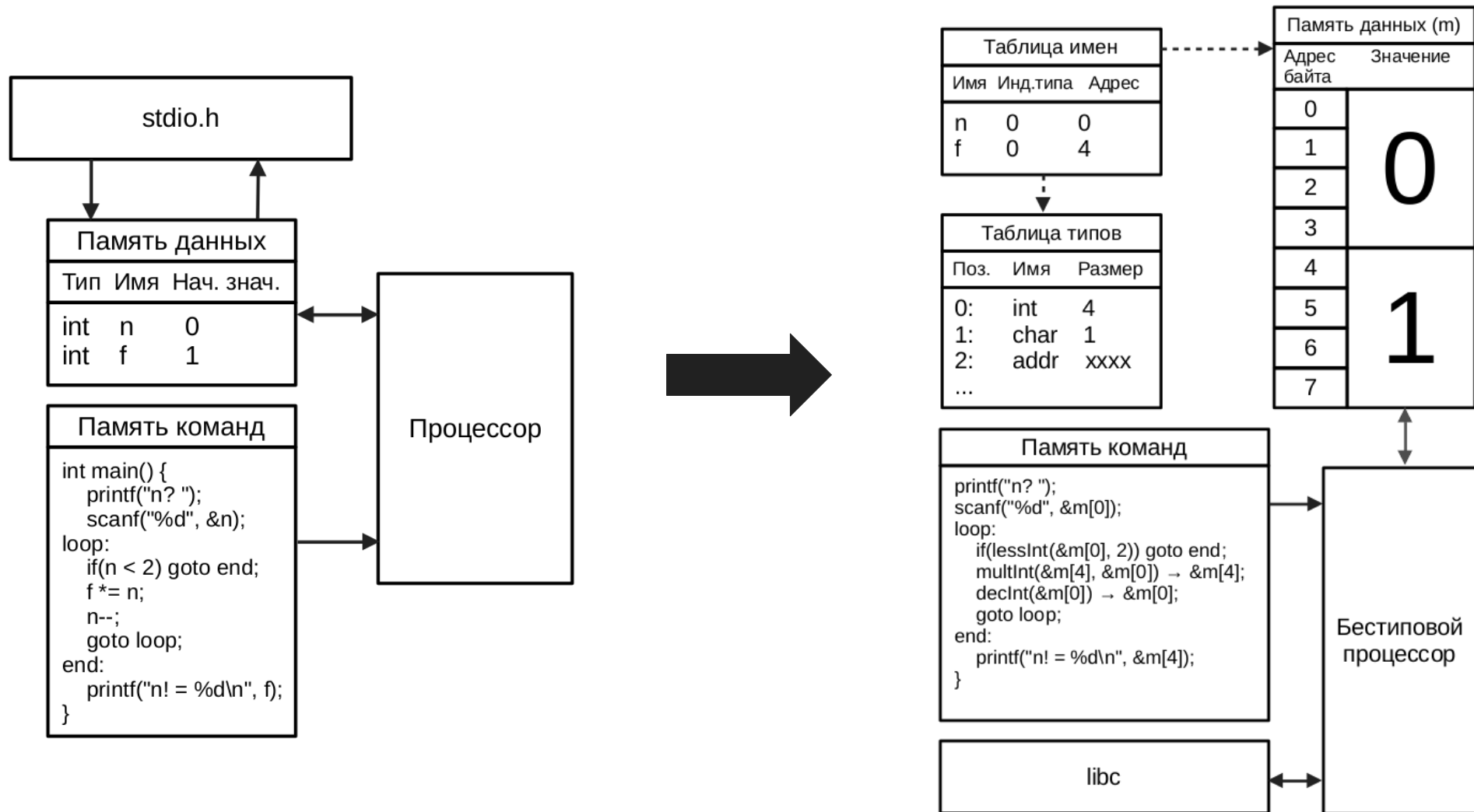
lessInt(void*, void*) → void*

multInt(void*, void*) → void*

movInt(void*) → void*

decInt(void*) → void*

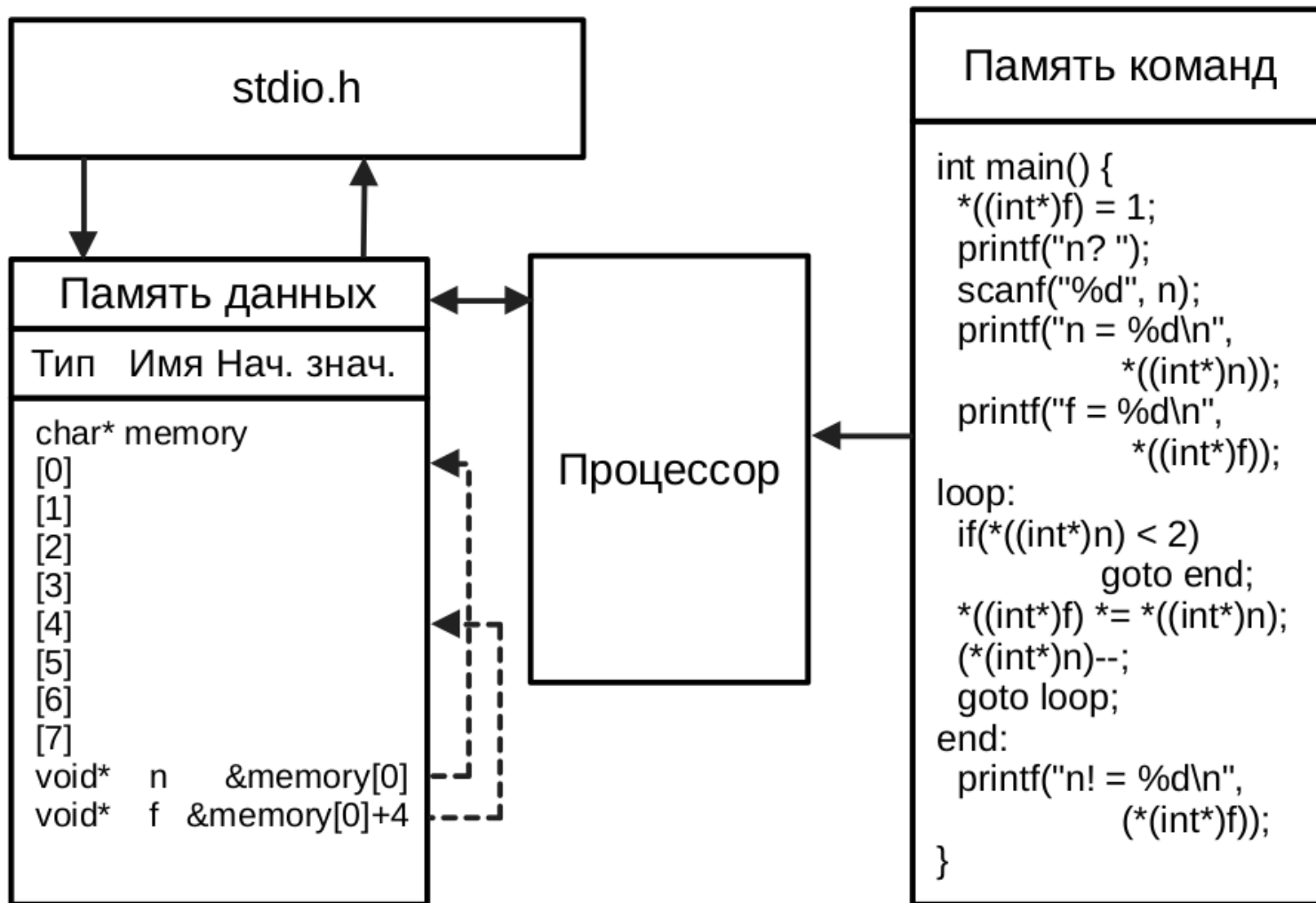
Трансформация статической типизации



Бестиповое программирование на C

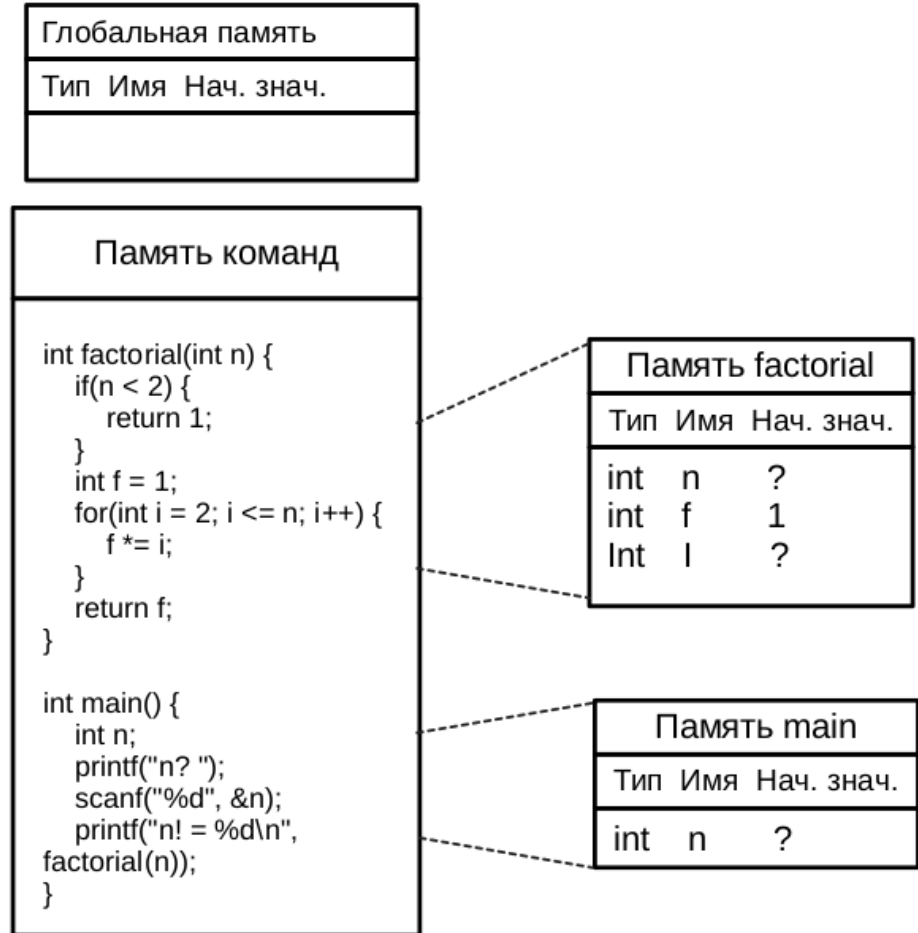
```
1 #include <stdio.h>
2
3 static char memory[2*sizeof(int)]; // Память для n и f
4 static void* n = memory; // Адрес на область для n
5 static void* f = memory + sizeof(int); // Адрес на область для f
6
7 int main() {
8     *((int*)f) = 1;
9     printf("n? ");
10    scanf("%d", &n);
11    printf("n = %d\n", *((int*)n));
12    printf("f = %d\n", *((int*)f));
13 loop:
14     if(*((int*)n) < 2) goto end;
15     *((int*)f) *= *((int*)n);
16     (*(int*)n)--;
17     goto loop;
18 end:
19     printf("n! = %d\n", (*(int*)f));
20     return 0;
21 }
```

Отображение бестиповой программы на структуру



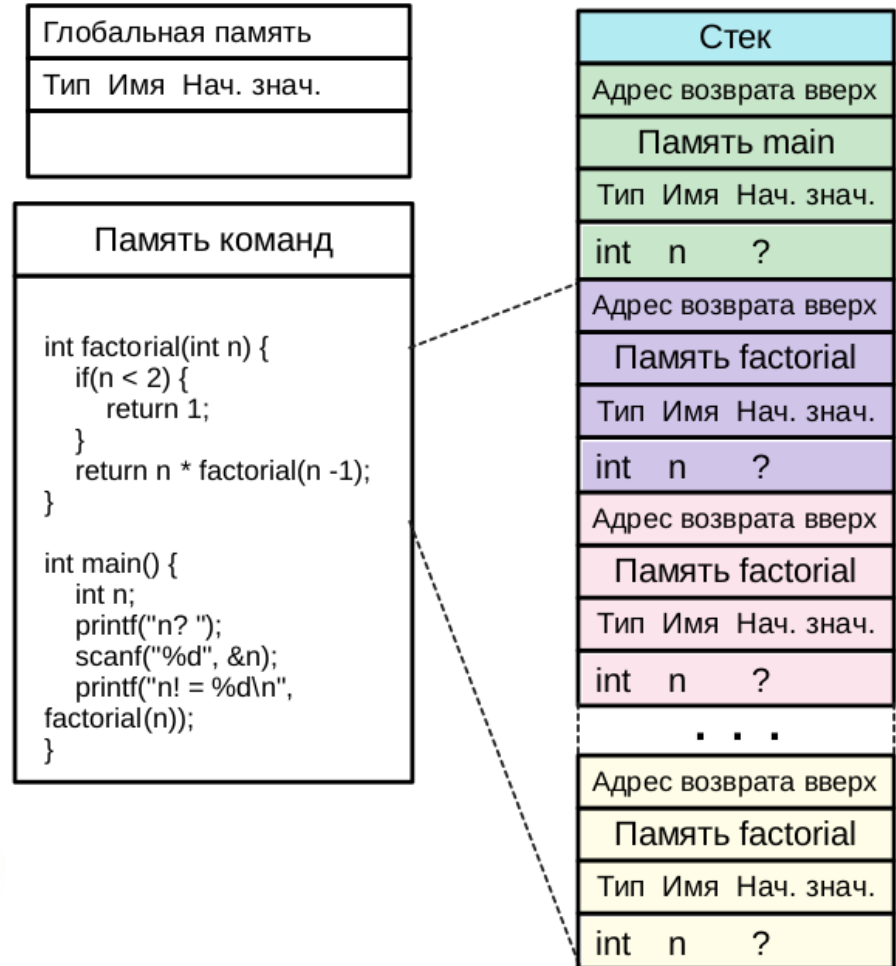
Статическая типизация и локальные данные

```
1 #include <stdio.h>
2
3 int factorial(int n) {
4     if(n < 2) {
5         return 1;
6     }
7     int f = 1;
8     for(int i = 2; i <= n; i++) {
9         f *= i;
10    }
11    return f;
12 }
13
14 int main() {
15     int n;
16     printf("n? ");
17     scanf("%d", &n);
18     printf("n! = %d\n", factorial(n));
19 }
```



Статическая типизация и рекурсия

```
1 #include <stdio.h>
2
3 int factorial(int n) {
4     if(n < 2) {
5         return 1;
6     }
7     return n * factorial(n - 1);
8 }
9
10 int main() {
11     int n;
12     printf("n? ");
13     scanf("%d", &n);
14     printf("n! = %d\n", factorial(n));
15 }
```



Список источников информации по данной теме

1. [Википедия] Динамическая типизация
https://ru.wikipedia.org/wiki/Динамическая_типизация
2. [Википедия] Статическая типизация
https://ru.wikipedia.org/wiki/Статическая_типизация
3. Статическая и динамическая типизация
<https://habr.com/ru/post/308484/>

Вопросы для обсуждения

1. Основная идея однозначности выполнения операций. Способы достижения однозначности.
2. Достоинства и недостатки операционной однозначности.
3. Достоинства и недостатки динамической однозначности.
4. Достоинства и недостатки статической однозначности.
5. Связь между однозначностью и методами типизации
6. Нужна ли динамическая проверка типов данных в статически типизированных языках?
7. Для чего в статически типизированных языках могут применяться бестиповые решения?
8. Когда в статически типизированных языках появляется необходимость динамической проверки типов?