Компилятор и его друзья

Андрей Комаров

11 июля 2016 г.

Знакомые компиляторы

Паскаль

```
fpc hello.pas
```

C

```
gcc hello.c -o hello
```

C++

```
g++ hello.cpp -o hello
```

Тоже компиляторы

Существенно отличаются от предыдущих!

Java

```
javac Hello.java
```

Python

```
python -m py_compile hello.py
```

Как работает компьютер?

- Компьютер очень глупый

Как работает компьютер?

- Компьютер очень глупый
  - Не понимает ни один язык программирования

Как работает компьютер?

- Компьютер очень глупый
  - Не понимает ни один язык программирования
  - Даже *язык ассемблера*

Как работает компьютер?

- Компьютер очень глупый
  - Не понимает ни один язык программирования
  - Даже *язык ассемблера*
- Всё, что он понимает — *машинный код*
  - Последовательность двоичных данных

Всё плохо!

- Разные устройства «говорят» на разных «языках»
  - Программу для компьютера нельзя запустить на телефоне или микроволновке и наоборот

Всё плохо!

- Разные устройства «говорят» на разных «языках»
  - Программу для компьютера нельзя запустить на телефоне или микроволновке и наоборот
- Всё ещё хуже: программы для одной операционной системы не работают на другой[1]

[1] В интересное время живём: Linux-программы недавно стало можно запускать в Windows 10

Что делать?

Проблема:

- Хочется писать на «человеческом» языке
- . . . но компьютер понимает только машинный код

Что делать?

Проблема:

- Хочется писать на «человеческом» языке
- . . . но компьютер понимает только машинный код

Решение:

- Написать «переводчик» с человеческого языка на компьютерный
- Компилятор — тот самый переводчик

Что делать?

Проблема:

- Хочется писать на «человеческом» языке
- . . . но компьютер понимает только машинный код

Решение:

- Написать «переводчик» с человеческого языка на компьютерный
- Компилятор — тот самый переводчик

Бонус-вопрос: на каком языке была написана самая первая программа?

Что происходит?

```c
#include <stdio.h>
int main() {
  printf("hello\n");
  return 0;
}
```

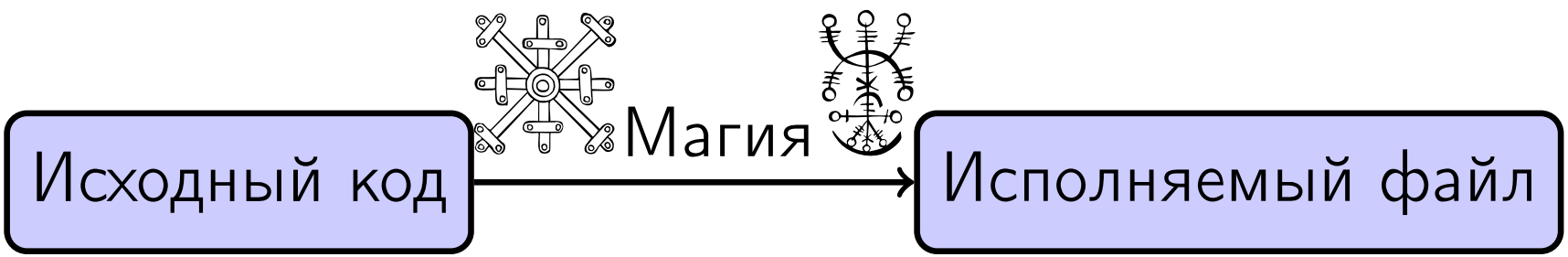
Компилируем:

```
gcc hello.c -o hello.exe
```

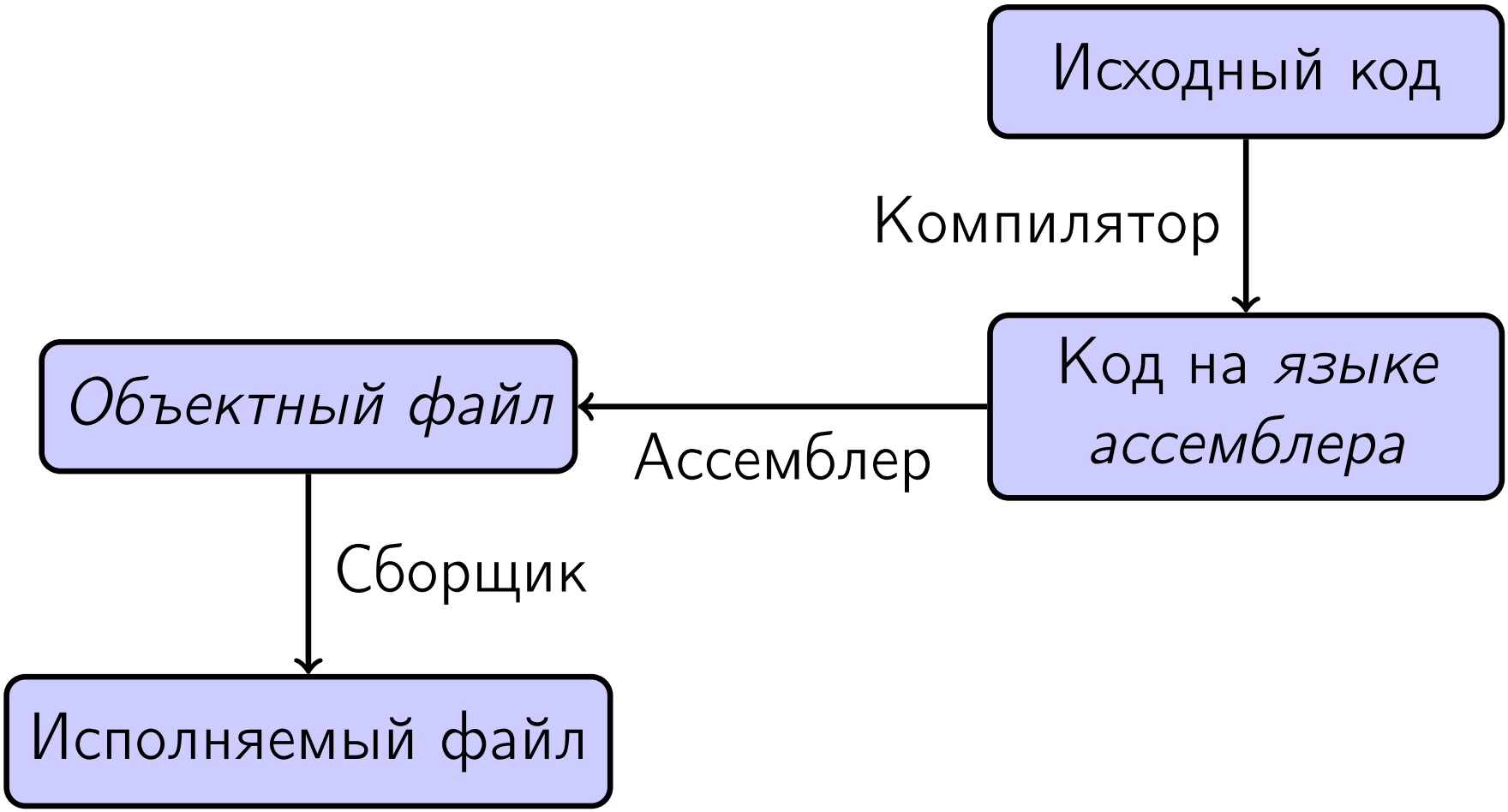
Запускаем:

```
./hello.exe
```

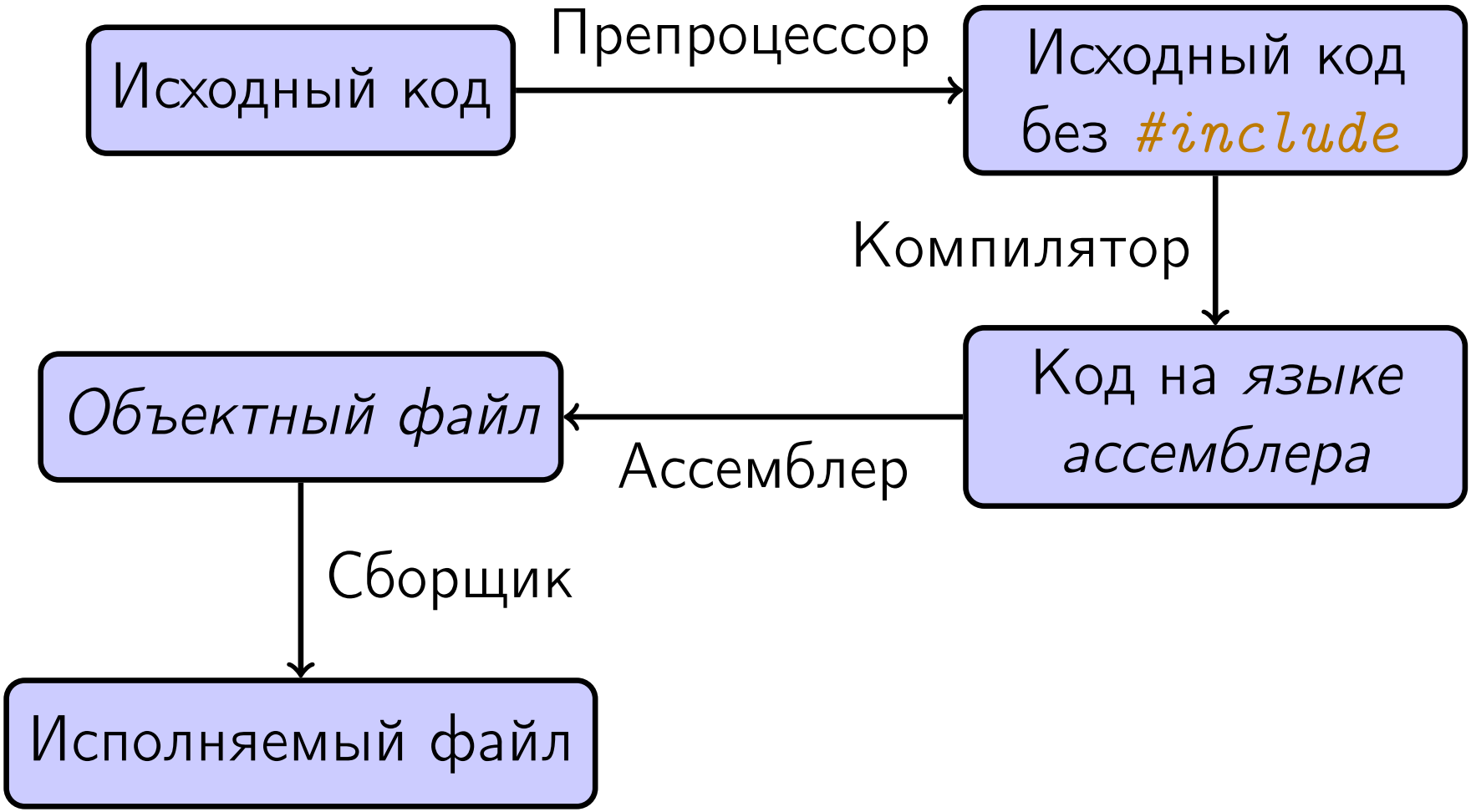
Фазы компиляции



Фазы компиляции Паскаля



Фазы компиляции Си



Препроцессор

Что делает:

- Раскрывает *#include*-ы
- Раскрывает *#define*-ы

Исследуем две программы.

C (hello.c):

```c
#include <stdio.h>
int main() {
  printf("hello\n");
  return 0;
}
```

C++ (hello.cpp):

```cpp
#include <iostream>
int main() {
  std::cout << "hello\n";
  return 0;
}
```

Препроцессор

`gcc -E hello.c -o hello-c.i`

Создался `hello-c.i`: 854 строки, 16860 байт[2]:

```
...
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;
...
extern int printf (const char *__restrict __format, ...);
...
int main() {
  printf("hello\n");
  return 0;
}
```

[2]Цифры на вашем компьютере могут существенно отличаться

Препроцессор

`g++ -E hello.cpp -o hello-cpp.ii`
Создался `hello-cpp.ii`: **27308** строки, **642284** байта:

```
...
namespace std
{
  typedef long unsigned int size_t;
  typedef long int ptrdiff_t;
  typedef decltype(nullptr) nullptr_t;
}
...
namespace std __attribute__ ((__visibility__ ("default")))
{
  extern istream cin;
  extern ostream cout;
  extern ostream cerr;
  ...
}
...
int main() {
  std::cout << "hello\n";
  return 0;
}
```

Препроцессор

Очень сложно, ничего не понятно!

```
gcc -E -c pre.c -o pre.i
```

pre.h:

```
void f();
```

pre.c:

```
#include "pre.h"
#define X (2 + 2)
const int a = X;
```

pre.i:

```
void f();
const int a = (2 +
```

Компилятор

- Преобразует код в код на *языке ассемблера*

Компилятор

- Преобразует код в код на *языке ассемблера*

Что такое язык ассемблера?

- Человекочитаемая запись (*мнемоники*) машинного кода
  - Всё ещё не понятна компьютеру
- Свой для каждой *архитектуры*
  - Компьютер (x86-64)
  - Телефон (ARM)
  - Микроволновка (AVR)

Компилятор

gcc -S hello.c -o hello.s

```
        .file   "hello.c"
        .section        .rodata
.LC0:
        .string "hello"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
        call    puts
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (GNU) 6.1.1 20160501"
        .section        .note.GNU-stack,"",@progbits
```

Компилятор

Очистим от «мусора»:

```
        .section        .rodata
.LC0:
        .string         "hello"
        .text
        .globl          main
main:
        pushq           %rbp
        movq            %rsp, %rbp
        movl            $.LC0, %edi
        call            puts
        movl            $0, %eax
        popq            %rbp
        ret
```

Ассемблер

- Преобразует код на языке ассемблера в *объектные файлы*

Ассемблер

- Преобразует код на языке ассемблера в *объектные файлы*

Что такое объектные файлы?

- Файлы с *машинным кодом* и дополнительной *полезной информацией*

Ассемблер

- Преобразует код на языке ассемблера в *объектные файлы*

Что такое объектные файлы?

- Файлы с *машинным кодом* и дополнительной *полезной информацией*

Что ещё за полезная информация?

Объектные файлы

- Есть большой проект
- Поменяли один файл и не хотим ждать час, пока пересоберётся

Объектные файлы

- Есть большой проект
- Поменяли один файл и не хотим ждать час, пока пересоберётся
- *Единицы трансляции*
  - Компилируем каждый файл отдельно
  - Объединяем результаты
    - Это быстрее, чем перекомпилировать

Объектные файлы

a.c:

```
void f(char*);
int x = 5;
int main() {
  f("SIS");
  x = 4; f("SIS");
  return 0;
}
```

b.c:

```
#include <stdio.h>
extern int x;
void f(char *name) {
  printf("Hi, %s, %d\n",
    name, x);
}
```

Компилируем:

```
gcc -c a.c -o a.o
gcc -c b.c -o b.o
gcc -s a.o b.o -o main
```

Объектные файлы

a.c:

```
void f(char*);
int x = 5;
int main() {
  f("SIS");
  x = 4; f("SIS");
  return 0;
}
```

b.c:

```
#include <stdio.h>
extern int x;
void f(char *name) {
  printf("Hi, %s, %d\n",
    name, x);
}
```

В каждом объектном файле есть:

- Список функций и переменных в нём
- Список функций и переменных, которые ему дополнительно нужны

Таблица символов:

nm a.o:

```
                 U f
0000000000000000 T main
0000000000000000 D x
```

nm b.o:

```
0000000000000000 T f
                 U printf
                 U x
```

Сборщик

Другие имена: компоновщик, линковщик, ...

- Собирает воедино несколько объектных файлов
- Проверяет, что во всех объектных файлах заткнуты все дырки
- Создаёт исполняемый файл, который уже можно запустить

```
gcc -c a.c -o a.o
gcc -c b.c -o b.o
gcc -s a.o b.o -o main # <-- Вот он
```

Стандартная библиотека

Обман! `printf` не объявлен ни в `a.o`, ни в `b.o`, но `b.o` его требует!

```
gcc -c a.c -o a.o
gcc -c b.c -o b.o
gcc -s a.o b.o -o main # <-- Обман!
```

Таблица символов:

nm a.o:

```
                 U f
0000000000000000 T main
0000000000000000 D x
```

nm b.o:

```
0000000000000000 T f
                 U printf
                 U x
```

Стандартная библиотека

- `printf` — функция из *стандартной библиотеки C*
- Объектный файл стандартной библиотеки по умолчанию добавляется сборщиком

Команда	**Размер**[3]
gcc hello.c -o hello	6728
g++ hello.cpp -o hello	8080
gcc -static hello.c -o hello	804280
g++ -static hello.cpp -o hello	2056696
fpc hello.pas -ohello	176416

[3] У вас может отличаться

Статическая и динамическая линковка

- Не в этот раз
- Большая тема для отдельной лекции
- Вкратце:
  - Статическая — запихать всё в исполняемый файл
    - Здоровенный исполняемый файл
  - Динамическая — дать подсказку, в каком файле и где найти нужную функцию
    - Не здоровенный исполняемый файл

Бонус 1. Хак Томпсона

- Есть программа с открытым исходным кодом
- Вы прочитали этот код и не нашли ничего криминального
- Можно ли ему *доверять* (запускать на своей системе)?

Бонус 1. Хак Томпсона

- Есть программа с открытым исходным кодом
- Вы прочитали этот код и не нашли ничего криминального
- Можно ли ему *доверять* (запускать на своей системе)?

НЕТ

Хак Томпсона, попытка 1

Основная идея — скомпилированные программы никто не читает (это долго и мерзко)

- Есть компилятор (назовём его gcc) и его исходный код gcc.c
- Важная системная программа^{TM}(назовём её login) и её исходный код login.c
- gcc(login.c) = login
- Напишем плохой login'
- Напишем gcc'.c: gcc'(login.c) = login'
- Отдадим пользователю gcc.c, gcc', login.c и login'

Хак Томпсона, попытка 1

- У пользователя gcc.c, gcc', login.c и login'
- Пользователь не доверяет login' и решает его пересобрать
- gcc'(login.c) = login'

Хак Томпсона, попытка 1

- У пользователя gcc.c, gcc', login.c и login'
- Пользователь не доверяет login' и решает его пересобрать
- gcc'(login.c) = login'

Но не всё потеряно:

- gcc'(gcc.c) = gcc
- gcc(login.c) = login

Хак Томпсона, попытка 2

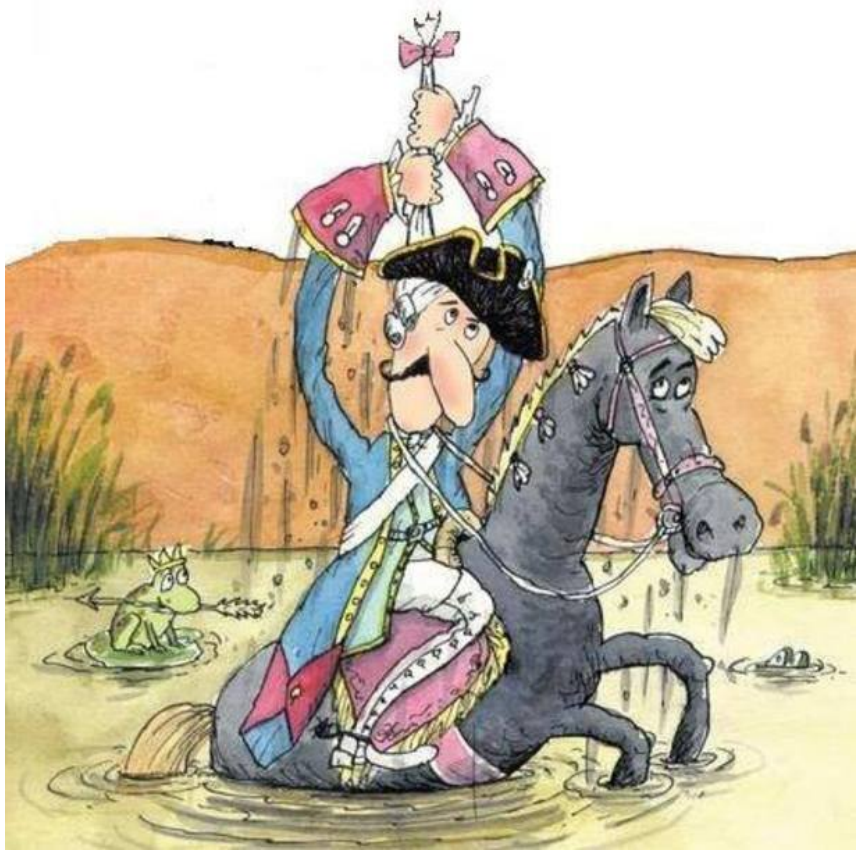
То же самое, но:

- gcc'(login.c) = login'
- **gcc'.c(gcc.c) = gcc'**

Мы все умрём. Решение: немножко доверять только тому, что написано лично

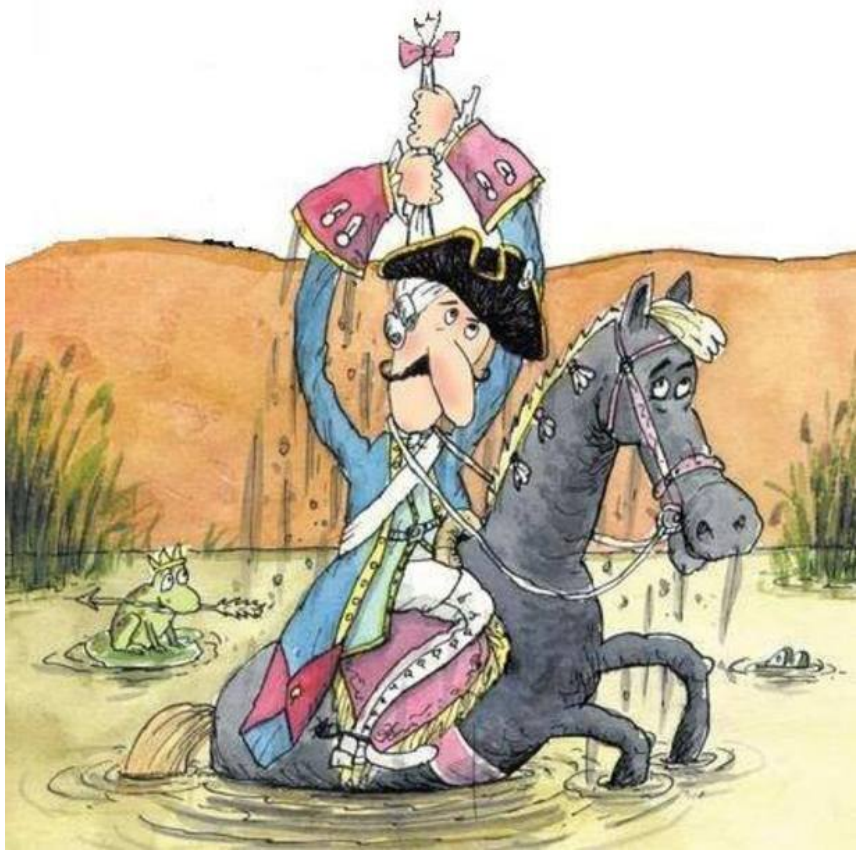
Бонус 2

- На каком языке была написана самая первая программа?



Бонус 2

- На каком языке была написана самая первая программа?
- Говорят, что компилятор Си написан на Си...



Bootstrapping

Boot — ботинок, strap — ремешок, петелька

Bootstrapping

- Компилятор языка X написан на языке X
- Но как его скомпилировали после того, как написали?

Bootstrapping, простой способ

Простой способ

1. Пишем на языке Y компилятор языка X (C_{YX})
2. Пишем на языке X компилятор языка X (C_{XX})
3. Компилируем C_{YX}
4. С помощью C_{YX} компилируем C_{XX}
5. С помощью C_{XX} компилируем C_{XX}

Bootstrapping, простой способ

Простой способ

1. Пишем на языке Y компилятор языка X (C_{YX})
2. Пишем на языке X компилятор языка X (C_{XX})
3. Компилируем C_{YX}
4. С помощью C_{YX} компилируем C_{XX}
5. С помощью C_{XX} компилируем C_{XX}

Проблема:

- Делать одно и то же два раза

Bootstrapping, сложный способ

Сложный способ

1. Пишем на языке Y компилятор *подмножества* языка X
2. Пишем на этом же *подмножестве* языка X компилятор полного языка X
3. Компилируем C_{YX}
4. С помощью C_{YX} компилируем C_{XX}
5. С помощью C_{XX} компилируем C_{XX}

Bootstrapping, сложный способ

Сложный способ

1. Пишем на языке Y компилятор *подмножества* языка X
2. Пишем на этом же *подмножестве* языка X компилятор полного языка X
3. Компилируем C_{YX}
4. С помощью C_{YX} компилируем C_{XX}
5. С помощью C_{XX} компилируем C_{XX}

Проблема:

- Не используется вся мощь X

Bootstrapping, ещё способ

Ещё способ

1. Пишем на языке Y компилятор *подмножества* X_1 языка X
2. Пишем на X_1 компилятор подмножества X_2 языка X
3. ...
4. ??????
5. Компилятор!

Вопросы?