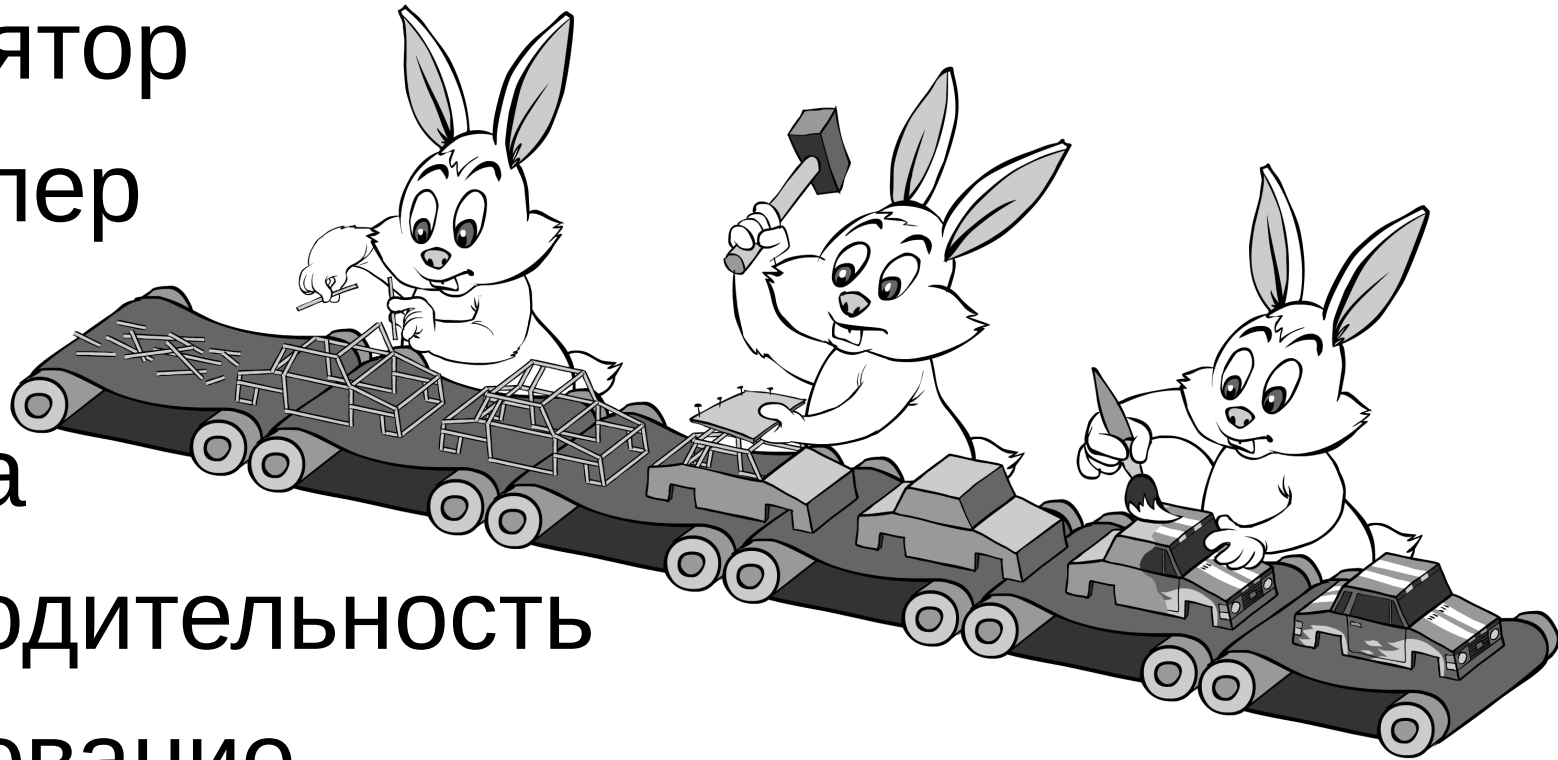


# Toolchain

- Системы компиляции
- Компилятор
- Ассемблер
- Линкер
- Отладка
- Производительность
- Портирование



# Как зовут преподавателя?

- Владимир Владимирович Игоревич
- Телефон: +7-903-842-27-55
- Email: [konstantin.vladimirov@gmail.com](mailto:konstantin.vladimirov@gmail.com)
- Где скачать слайды и конспект?  
[sourceforge.net/projects/cpp-lects-rus](https://sourceforge.net/projects/cpp-lects-rus)
- Помощь в редактировании и исправлении ошибок приветствуется

# Системы компиляции

- Более простое название toolchain (тулчейн)
- Набор средств для получения исполняемого файла из исходного кода



GNU Toolchain



LLVM Toolchain

# Системы компиляции

```
$ gcc -O2 program.c -o program.x
```



Что при этом происходит?



# Системы компиляции

```
$ gcc -O2 program.c -o program.x
```



Что при этом происходит?

- **Вызов компилятора**
- **Вызов ассемблера**
- **Вызов линкера**



# GCC: как посмотреть что в действительности запустилось?

```
$ gcc -O2 program.c --verbose -save-temps
```

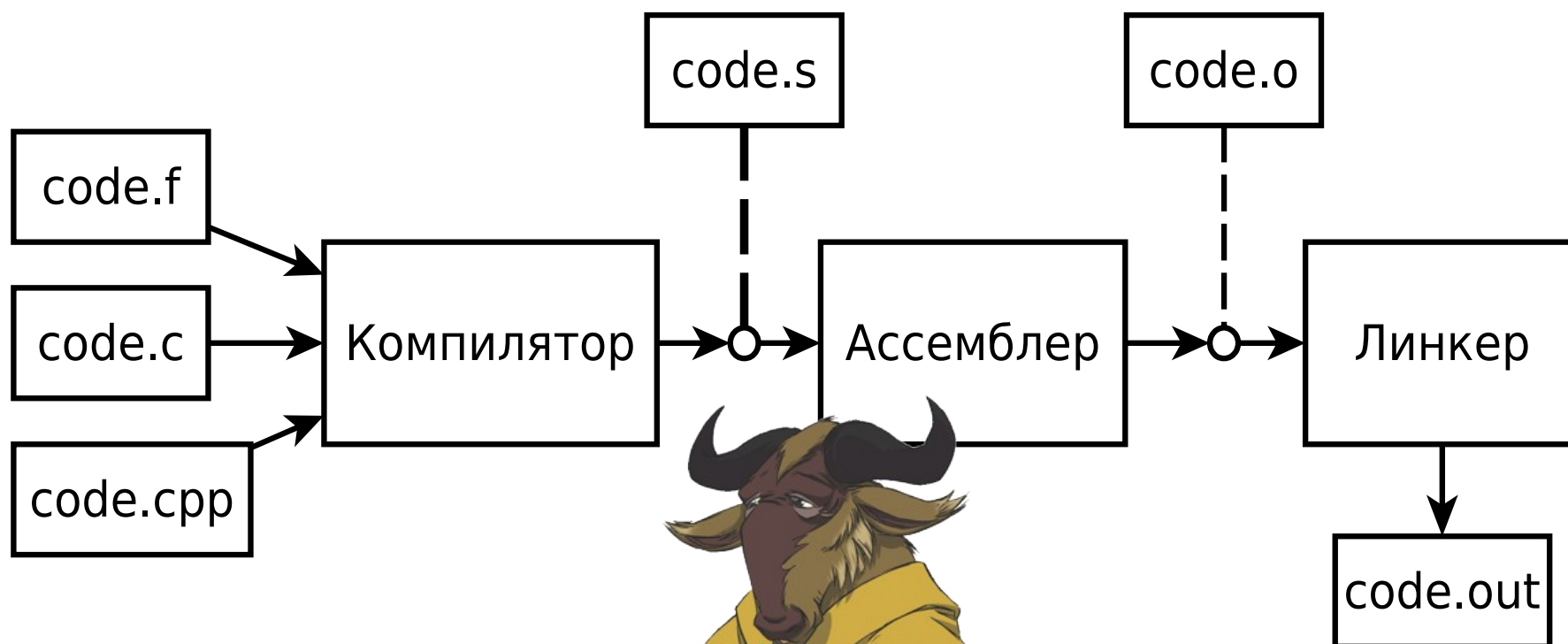
```
cc1 -E -quiet -v -imultiarch x86_64-linux-gnu program.c  
-mtune=generic -march=x86-64 -fpch-preprocess -o  
program.i
```

```
cc1 -fpreprocessed program.i -quiet -dumpbase program.c  
-mtune=generic -march=x86-64 -auxbase program -version -o  
program.s
```

```
as -v --64 -o program.o program.s
```

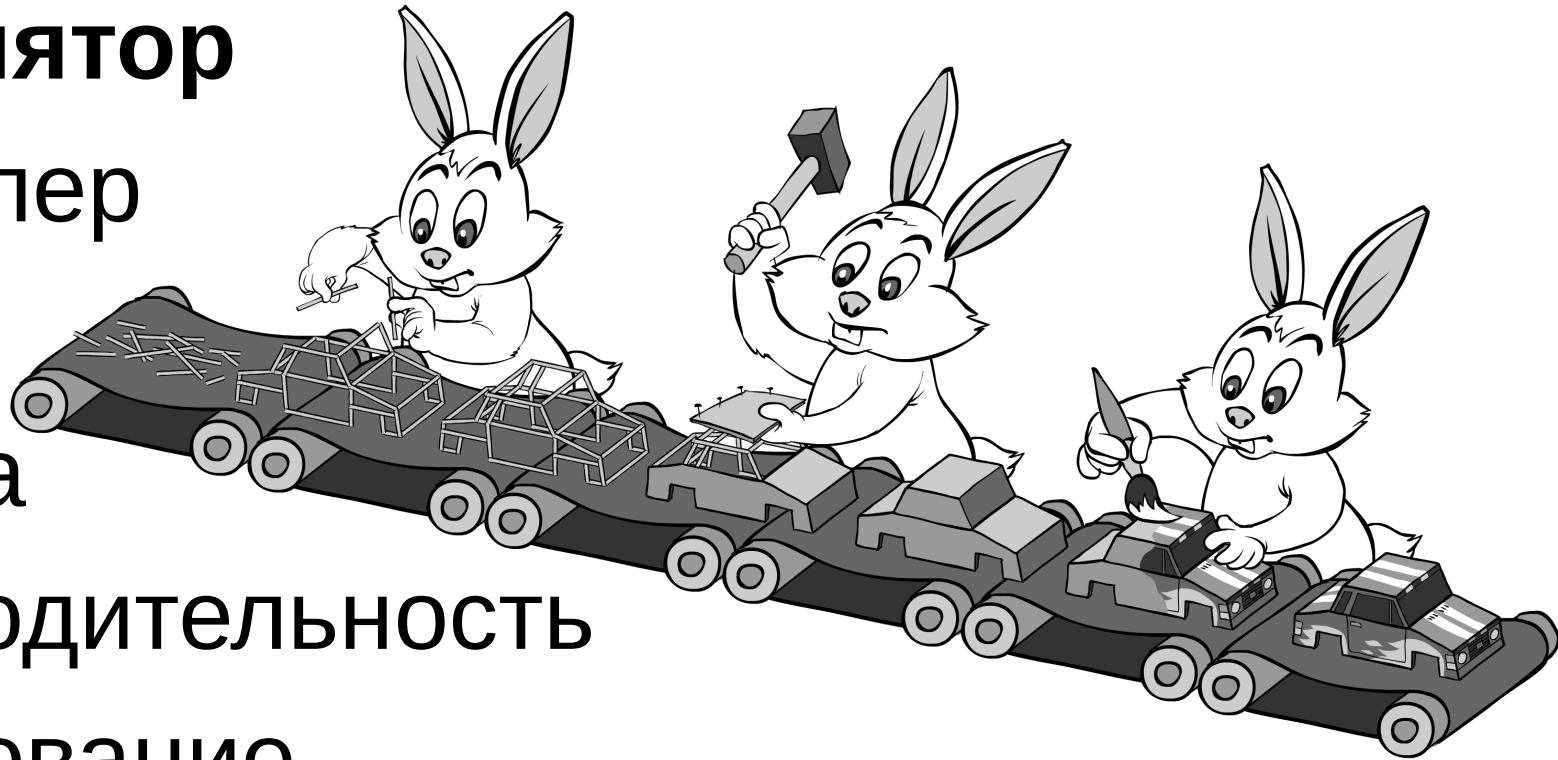
```
collect2 <....> program.o -lgcc --as-needed -lgcc_s --no-  
as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed  
crtn.o
```

# Упрощённая схема toolchain



# Toolchain

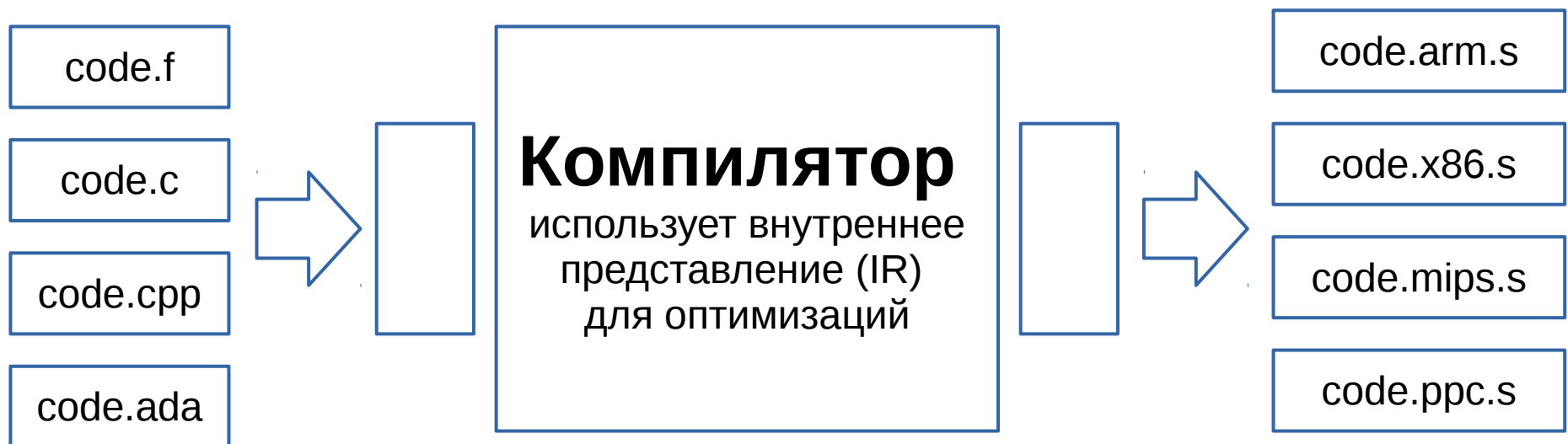
- Системы компиляции
- **Компилятор**
- Ассемблер
- Линкер
- Отладка
- Производительность
- Портирование





# Трансляция программы

- Вход: программа на одном из языков программирования
- Выход: ассемблер одной из архитектур
- Основная идея: отвязать компилятор от конкретного языка и ассемблера



# Компилятор с птичьего полёта



## Frontend

Препроцессинг  
Лексический анализ  
Синтаксический анализ  
Семантический анализ  
Построение HIR

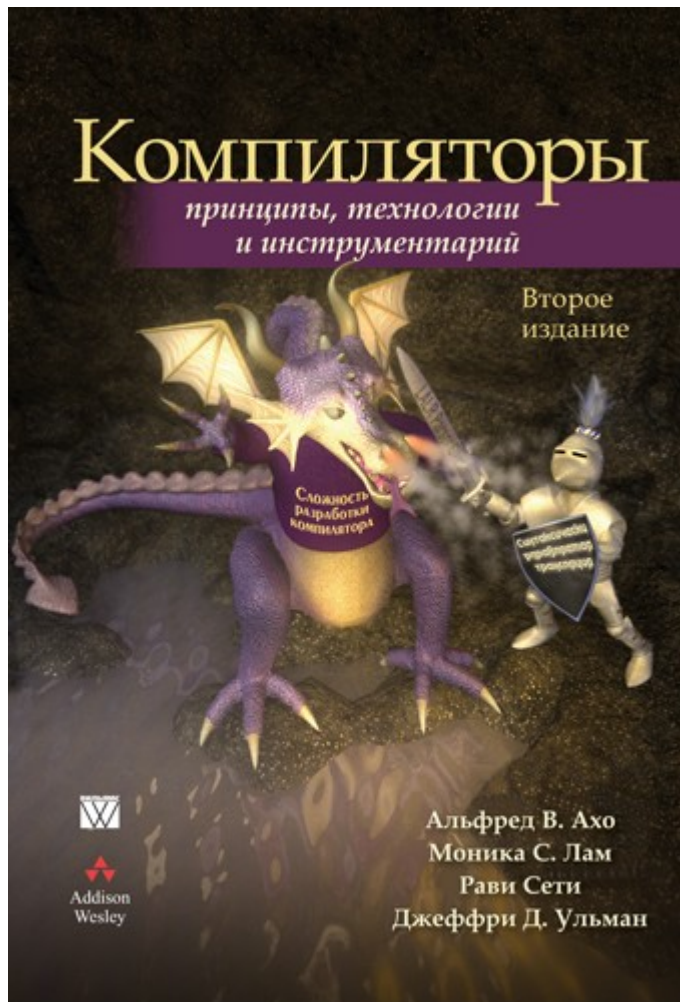


## Middleend + Backend

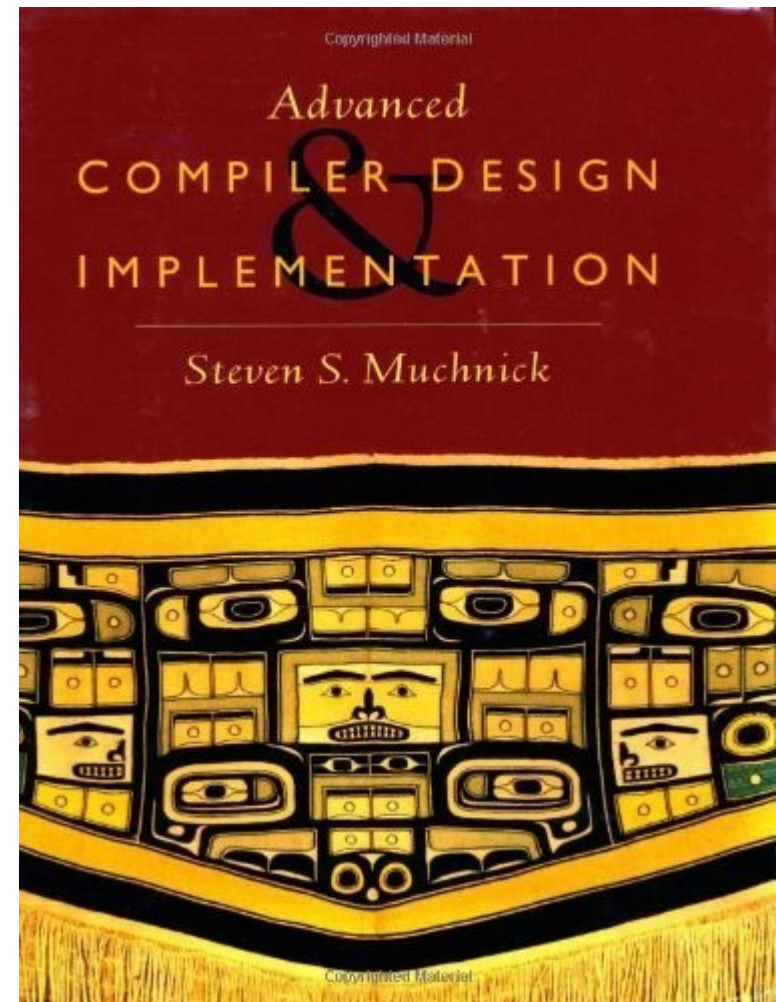
Оптимизации HIR  
Оптимизации MIR  
Оптимизации LIR  
Кодогенерация

# Литература

Frontend



Middleend + Backend



# Лексический анализ

- Ключевое понятие: лексема или токен
- Токен это пара тип/значение
- Исходный текст преобразуется в набор токенов

```
if( x > 3.1 ) { fprintf(stderr, "%s\n", "x too big"); abort(); }
```



keyword
if

bracket
(

operator
>

value
3.1

bracket
)



# Препроцессинг

- Препроцессингом называется обработка исходного текста программы, до её синтаксического анализа
- Наивно думать, что препроцессинг происходит до лексического анализа. На самом деле препроцессинг это фаза лексера

С птичьего полёта это  
было не особо заметно....



# Трансляция программы C++17

- Единица трансляции отображается в базовый набор символов, обрабатываются юникодные символы
- Конкатенируются строки, разбитые через \
- Комментарии заменяются на пробельные символы
- Файл разбивается на **препроцессинговые токены**
- Исполняются директивы препроцессора (include, define, прагмы)
- Заменяются escape-последовательности
- Соединяются строковые литералы
- Пробелы перестают иметь значение
- Препроцессинговые токены становятся **токенами**
- Проводится синтаксический анализ

# Изучение работы препроцессора

```
int foo (int a) {  
    // Will next line execute?\n    a += 1;  
    return a;  
}
```

- Произойдёт ли во время выполнения инкремент?

# Изучение работы препроцессора

```
int foo (int a) {  
    // Will next line execute?\n    a += 1;  
    return a;  
}
```



- Произойдёт ли во время выполнения инкремент?
- Нет, так как объединение строк предшествует удалению комментариев
- Но есть ли шансы узнать об этом заранее?



# Трансляция программы C++17

- Единица трансляции отображается в базовый набор символов, обрабатываются юникодные символы
  - Конкатенируются строки, разбитые через \
  - Комментарии заменяются на пробельные символы
  - Файл разбивается на **препроцессинговые токены**
  - Исполняются директивы препроцессора (include, define, прагмы)
- 
- Заменяются escape-последовательности
  - Соединяются строковые литералы
  - Пробелы перестают иметь значение
  - Препроцессинговые токены становятся **токенами**
  - Проводится синтаксический анализ

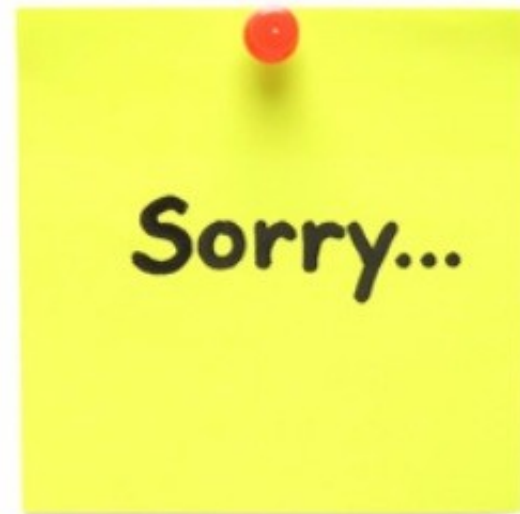


Отсюда можно  
сбросить результат

# Изучение работы препроцессора

```
$ gcc --std=c11 -E foo.c -o foo.i
```

```
# 1 "<command-line>" 2
# 1 "prep-magic.c"
int foo (int a) {
    return a;
}
```



- Ещё больше проблем могут доставить директивы препроцессора и макросы

# Изучение работы препроцессора

```
enum type_t {  
    OP_IGNORED = 0x0, OP_SRC = 0x1000,  
    OP_DST = 0x2000, OP_DSTSRC = 0x3000  
};  
  
bool is_op_src (enum type_t a) {  
    return (a & OP_SRC);  
}
```

```
enum type_t x = OP_DSTSRC;  
assert (is_op_src(x)); // FAILED!
```

# После препроцессирования

```
enum type_t {  
    OP_IGNORED = 0x0, OP_SRC = 0x1000,  
    OP_DST = 0x2000, OP_DSTSRC = 0x3000  
};  
  
char is_op_src (enum type_t a) {  
    return (a & OP_SRC);  
}
```

**myheader.h:**

#define bool char

// Happy debugging!

# Применения препроцессора

- **Использование по назначению**
  - Условное исключение кода из компиляции
  - Модульная структура программы, включение заголовочных файлов
  - Стражи включения
  - Стрингификация и конкатенация
- **Сомнительные применения**
  - Константы времени компиляции
  - Синонимы типов
  - Макросы (короткие обобщённые функции)

# Обсуждение

```
#include <iostream>
int main () { cout << "Hello" << endl; }
```

- Как вы думаете сколько в ней окажется строк после препроцессирования?
- А сколько байт будет размер у препроцессированного файла?

# Обсуждение

```
#include <iostream>
int main () { cout << "Hello" << endl; }
```

- Как вы думаете сколько в ней окажется строк после препроцессирования?
- А сколько байт будет размер у препроцессированного файла?

В моей реализации:

g++ 24706 loc, 704835 bytes

clang++ 37838 loc, 1004278 bytes

Вдумайтесь в эти результаты: мегабайт текста из одной строки.

# Предварительно собранные заголовочные файлы

- Традиционное расширение большинства компиляторов

```
$ g++ -c stdafx.h -o stdafx.h.gch
```

stdafx.h

```
#include <cassert>
#include <iostream>
#include <string>
#include <vector>
```



stdafx.h.gch

```
$ g++ -H mypch.cc
```

- Использует gch там где есть обычный include



# Проблема зависимостей

- Представьте, что вам надо решить что писать в Makefile как правило сборки

```
// x.c  
#include "inc.h"
```

```
x.o : x.c inc.h  
$(CC) x.c -o $@
```

- Но что, если inc.h сам по себе включает некий другой хедер?

```
// inc.h  
#include "inc2.h"
```

```
x.o : x.c inc.h inc2.h  
$(CC) x.c -o $@
```

# Проблема зависимостей

- Разумеется, сбор зависимостей следует автоматизировать (тоже в препроцессоре)

```
// x.c  
#include "inc.h"
```

```
// inc.h  
#include "inc2.h"
```

```
$ gcc -M user.c  
user.o: user.cc /usr/include/stdc-predef.h inc.h inc2.h  
$ gcc -MM user.c  
user.o: user.cc inc.h inc2.h  
$ gcc -MMD user.c
```

# Фронтенд

- Лексический анализ
- Синтаксический анализ
- Раскрытие шаблонов
- Вывод типов
- Построение AST



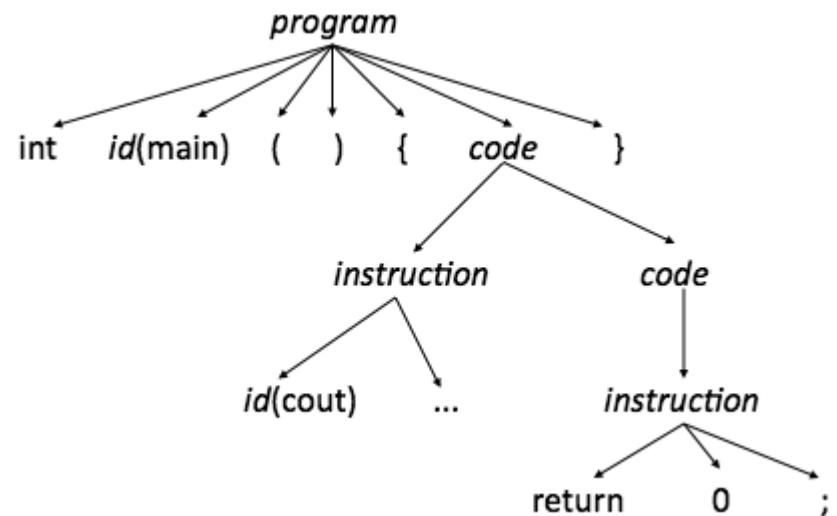
`C / C++ / Fortran / ...`



`Syntax tree (HIR)`

# Синтаксический анализ

- Синтаксический разбор
- Построение AST
- Разбор может быть нетривиальным (особенно в C++)

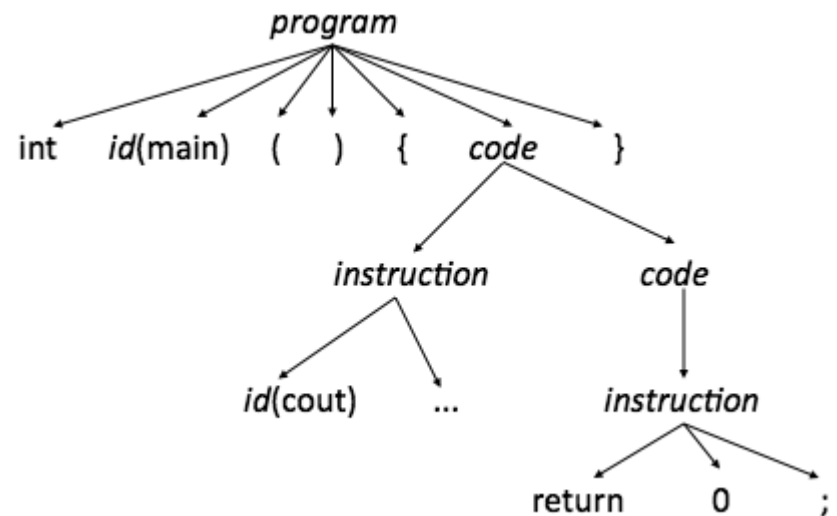


Пример. Что здесь написано?

```
ifstream datafile ("ins.dat");  
list<int> data (istream_iterator<int>(datafile),  
               istream_iterator<int>());
```

# Синтаксический анализ

- Синтаксический разбор
- Построение AST
- Разбор может быть нетривиальным (особенно в C++)



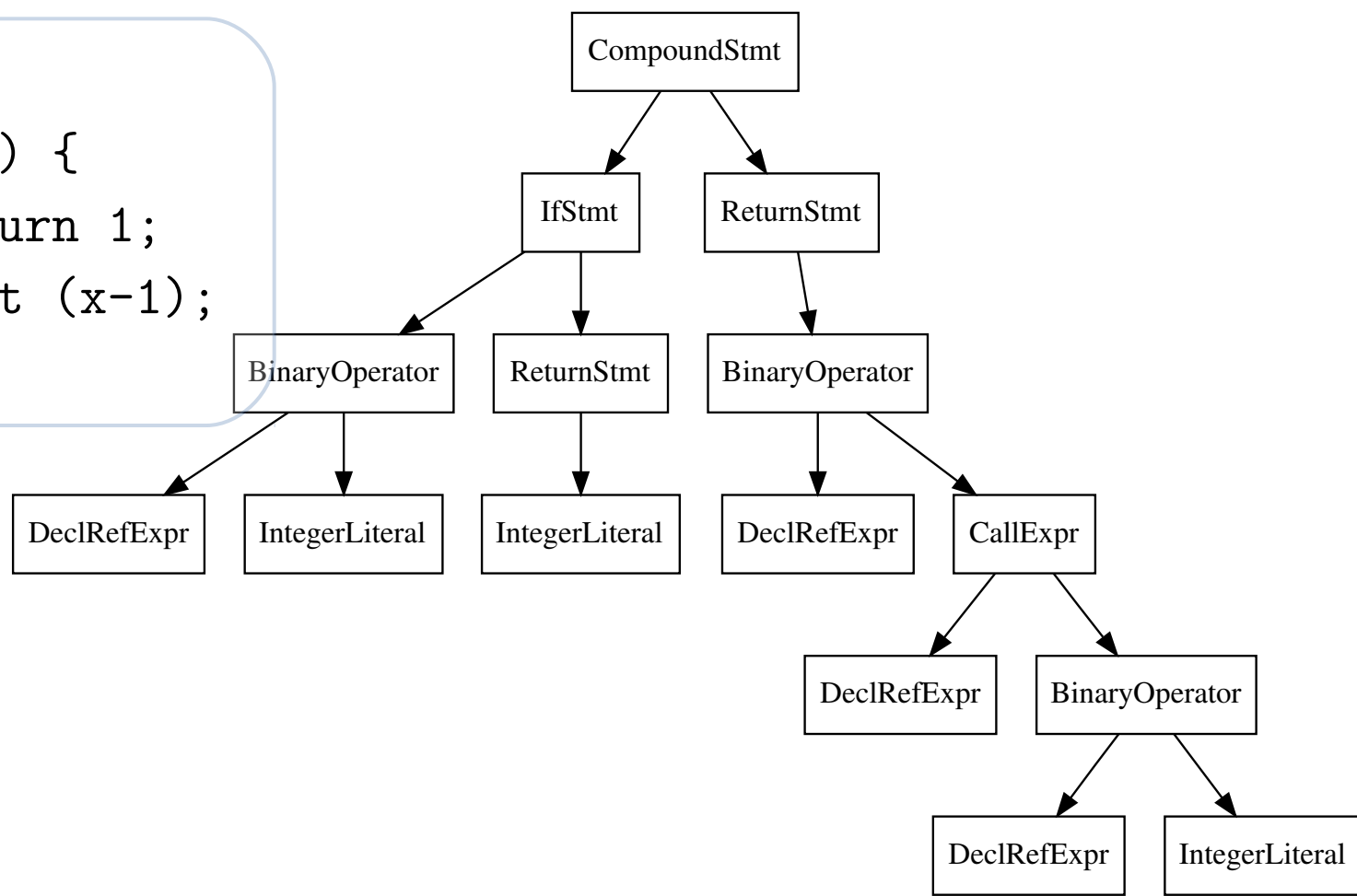
Пример 2. Что здесь написано?

```
template <class T> foo (T x) {  
    T::iterator *y;  
    /* .... */  
}
```

# Исследование AST

```
$ clang -cc1 -ast-view fact.c
```

```
unsigned  
fact (unsigned x) {  
    if (x < 2) return 1;  
    return x * fact (x-1);  
}
```

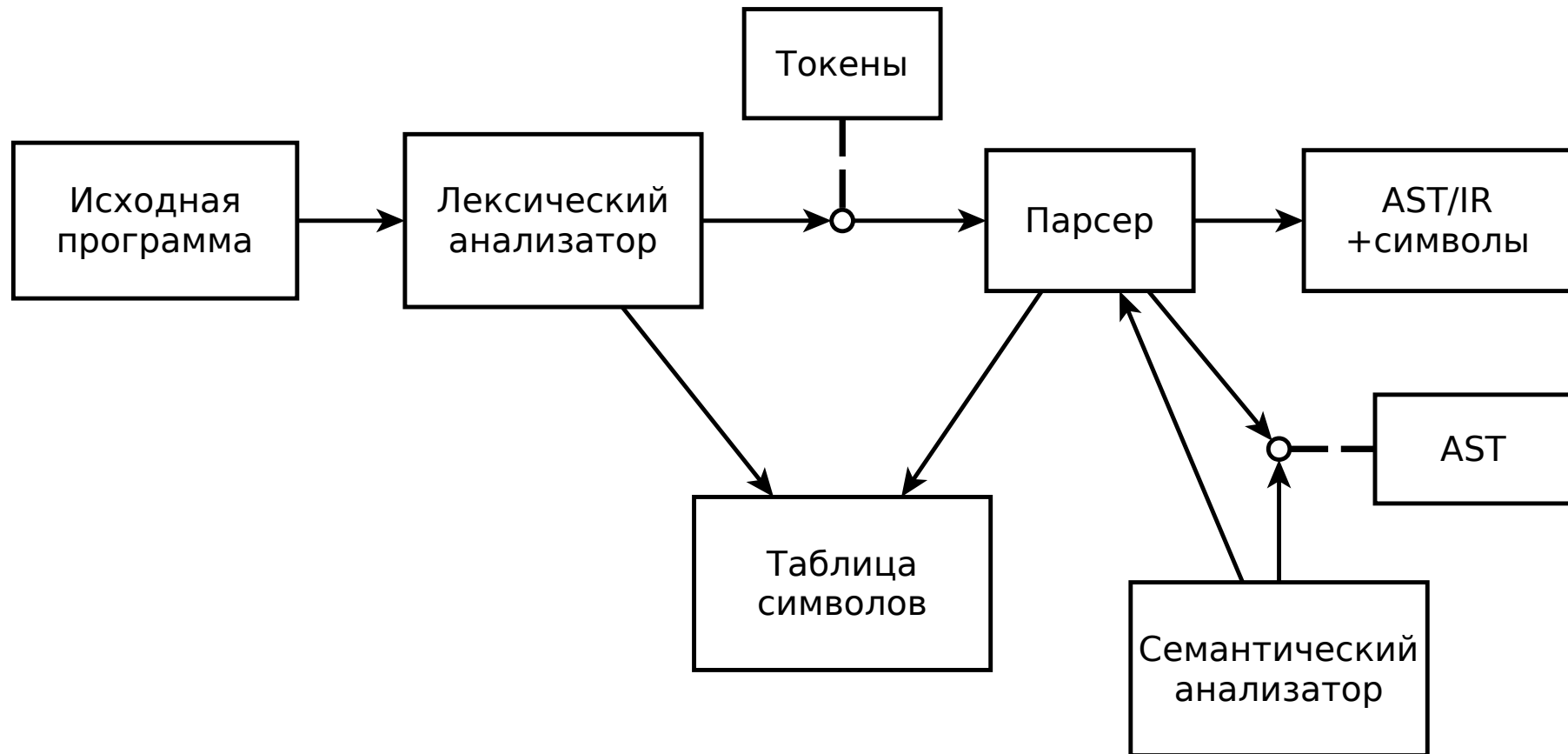


# Синтаксические макросы?

- В некоторых языках обработка до синтаксического анализа (препроцессор) заменена на возможность модифицировать сам синтаксический анализ
- Пример: синтаксические макросы в LISP
- Как по вашему, сделает ли это вещи существенно проще?
- Хотели бы вы сделать язык, где есть и то и другое?
- А сами писать на таком языке?
- А чтобы на нём писал ваш заклятый враг?
- Предположим, что вы ненавидите всё человечество. В этом случае как бы вы отнеслись к идее добавить синтаксические макросы в C++?

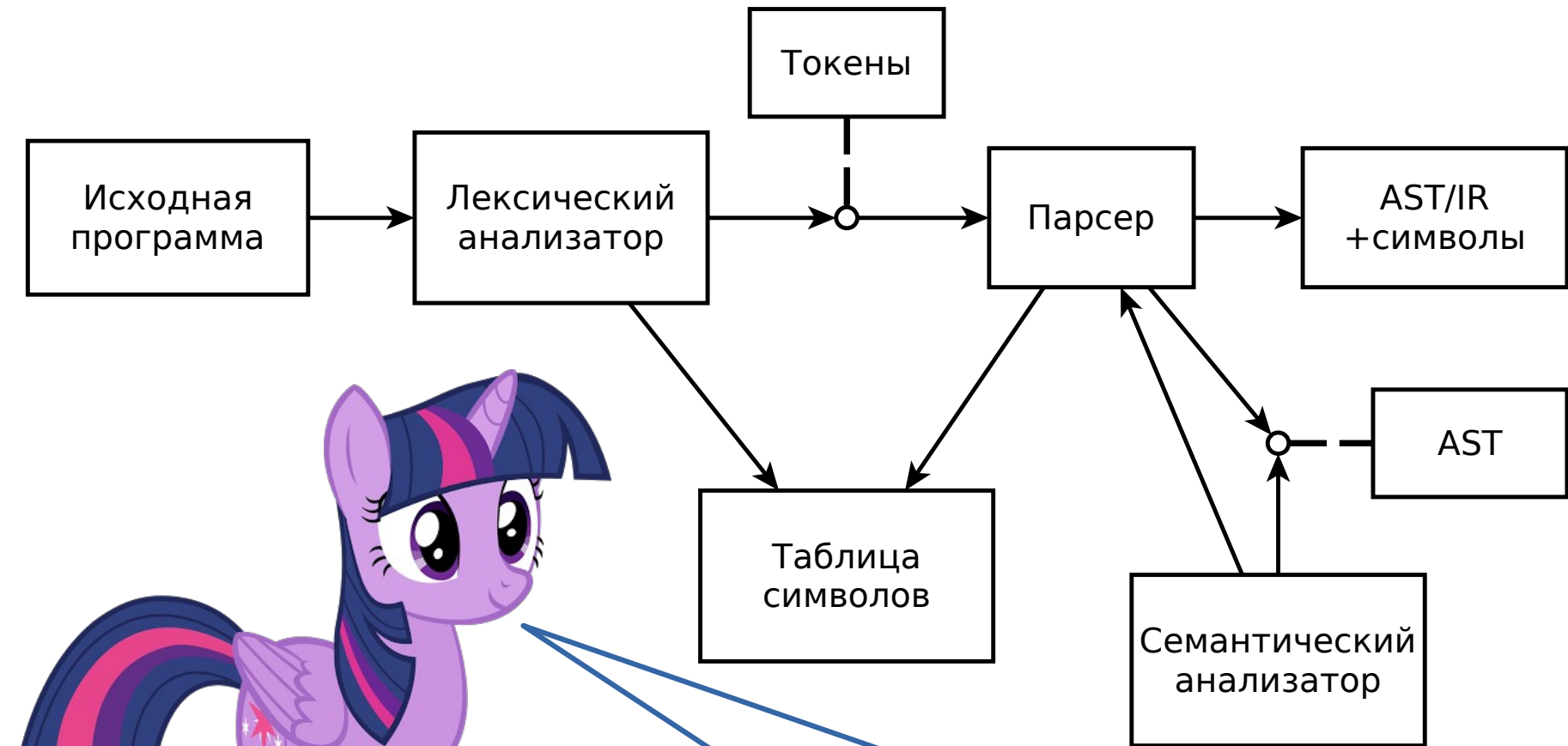


# Фронтенд (упрощённая схема)



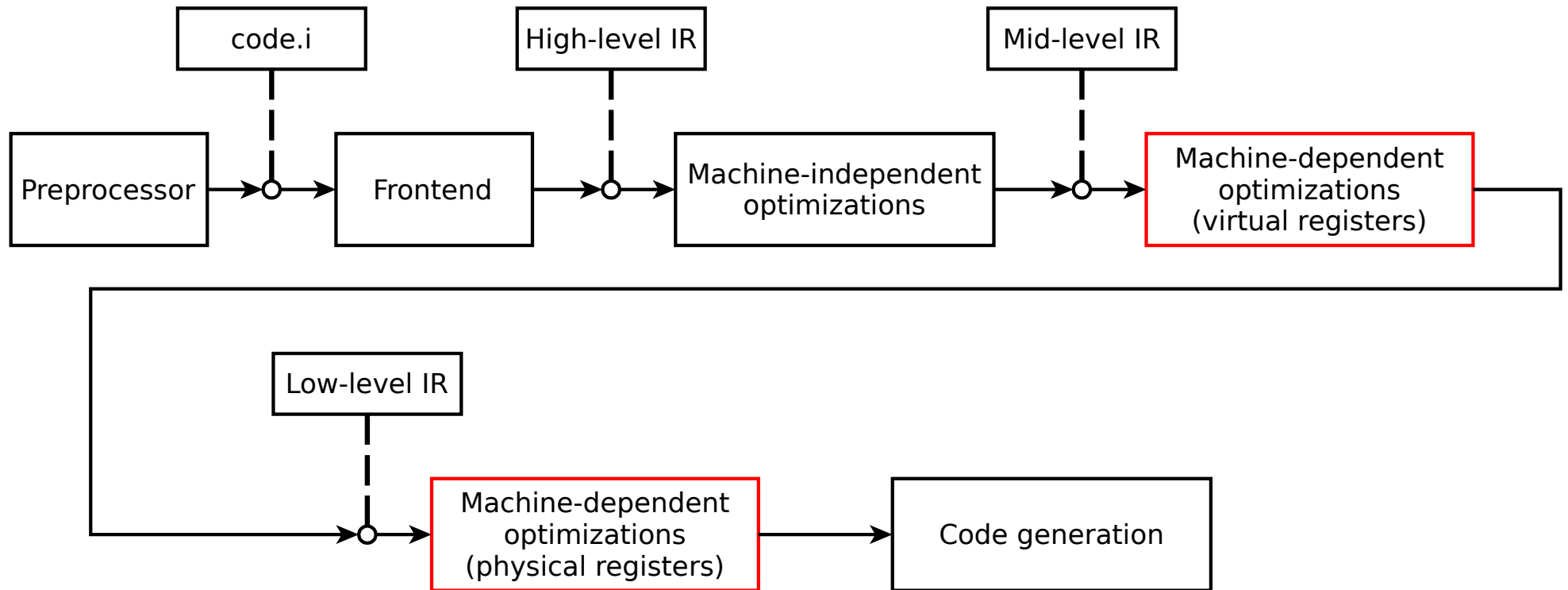


# Фронтенд (упрощённая схема)



Возможно это **слишком**  
упрощённая схема

# Что после фронтенда?



Syntax tree (HIR)



Virtual regs (MIR)



Physical regs (LIR)



Assembler code

# IR в GCC

GENERIC

RTL (virtual regs)

Gimple

RTL (physical regs)

Gimple SSA

Зачем их  
так много?



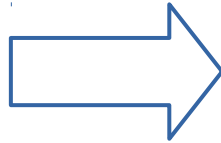
# SSA : static single assignment

```
x = 1;
```

```
y = x + 1;
```

```
x = 2;
```

```
z = x + 1;
```



```
x1 = 1;
```

```
y1 = x1 + 1;
```

```
x2 = 2;
```

```
z = x2 + 1;
```

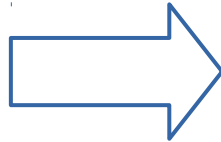
# SSA : static single assignment

```
x = 1;
```

```
y = x + 1;
```

```
x = 2;
```

```
z = x + 1;
```



```
x1 = 1;
```

```
y1 = x1 + 1;
```

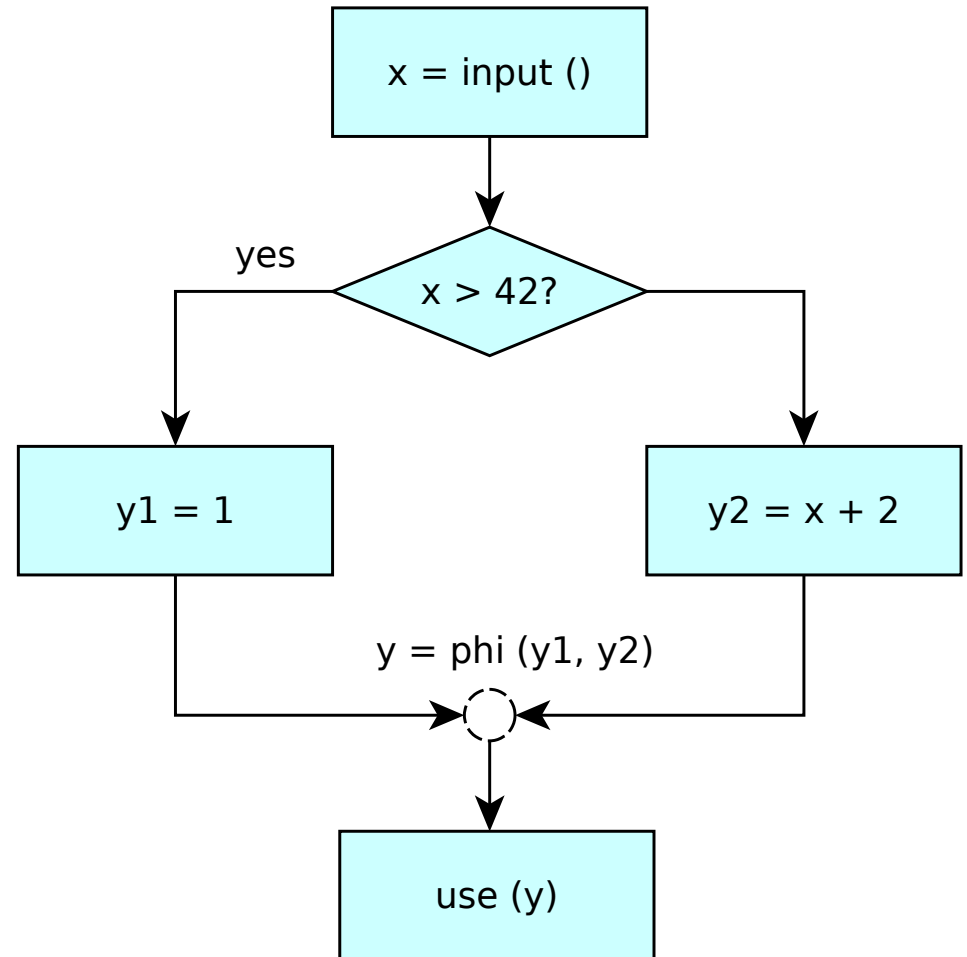
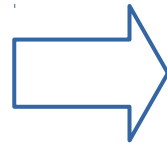
```
x2 = 2;
```

```
z = x2 + 1;
```

Проблема: как быть с  
ветвлениями и циклами?

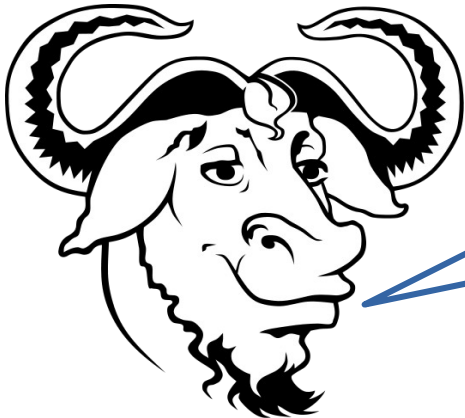
# SSA : static single assignment

```
x = input ();  
if (x > 42)  
    y = 1;  
else  
    y = x + 2;  
use (y);
```



# GCC: как получить дампы IR?

- `-fdump-tree-all-<options>`
- `-fdump-rtl-all-<options>`
- Можно получать графические дампы, если передать `<options> == graph`

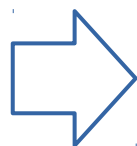


За полезными опциями всегда  
можно заглянуть сюда:  
<https://gcc.gnu.org/onlinedocs/gcc/>

# GCC: как читать gimple дампы

```
if (x < 2)
  return 1;

return
  x * fact (x-1);
```



```
<bb 3>:
  _3 = 1;
  goto <bb 5> (<L2>);

<bb 4>:
  _4 = x_2(D) + 4294967295;
  _5 = fact (_4);
  _6 = _5 * x_2(D);

<bb 5>:
  # _1 = PHI <_3(3), _6(4)>
<L2>:
  return _1;
```

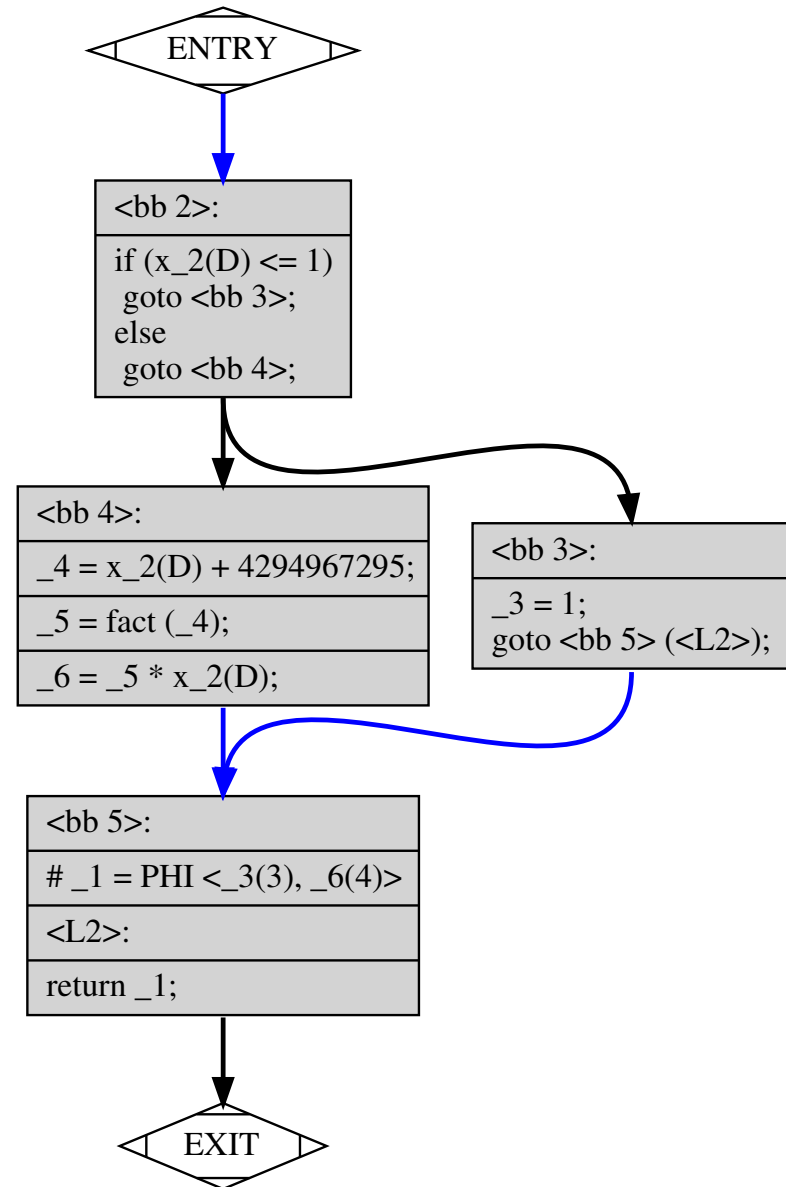




# Перевод в SSA форму

```
unsigned
fact (unsigned x)
{
    if (x < 2)
        return 1;

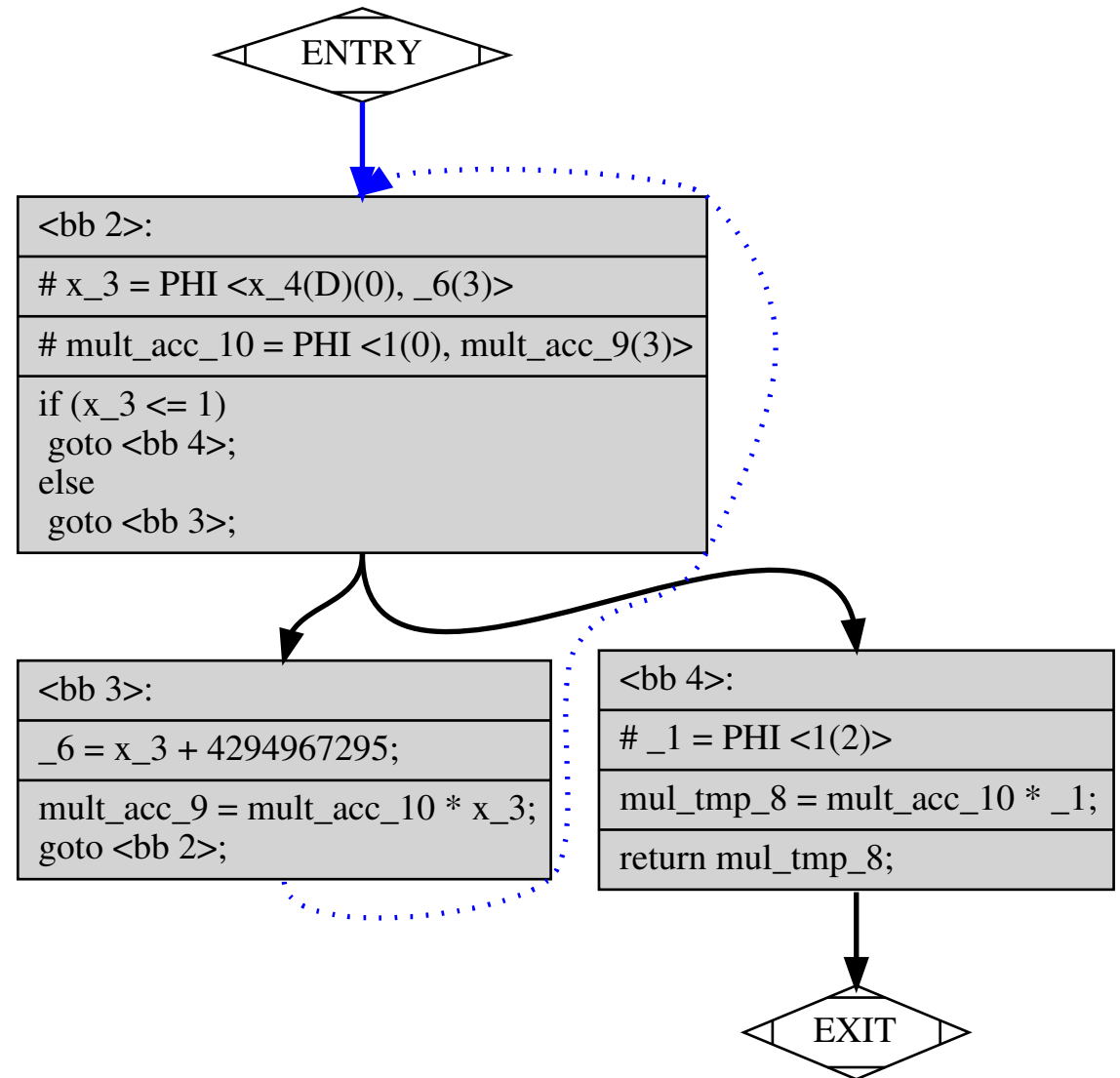
    return
        x * fact (x-1);
}
```



# Машинно-независимые преобразования

```
unsigned
fact (unsigned x)
{
    if (x < 2)
        return 1;

    return
        x * fact (x-1);
}
```



# GCC: как читать RTL дампы

`_6 = x_7 + 4294967295`

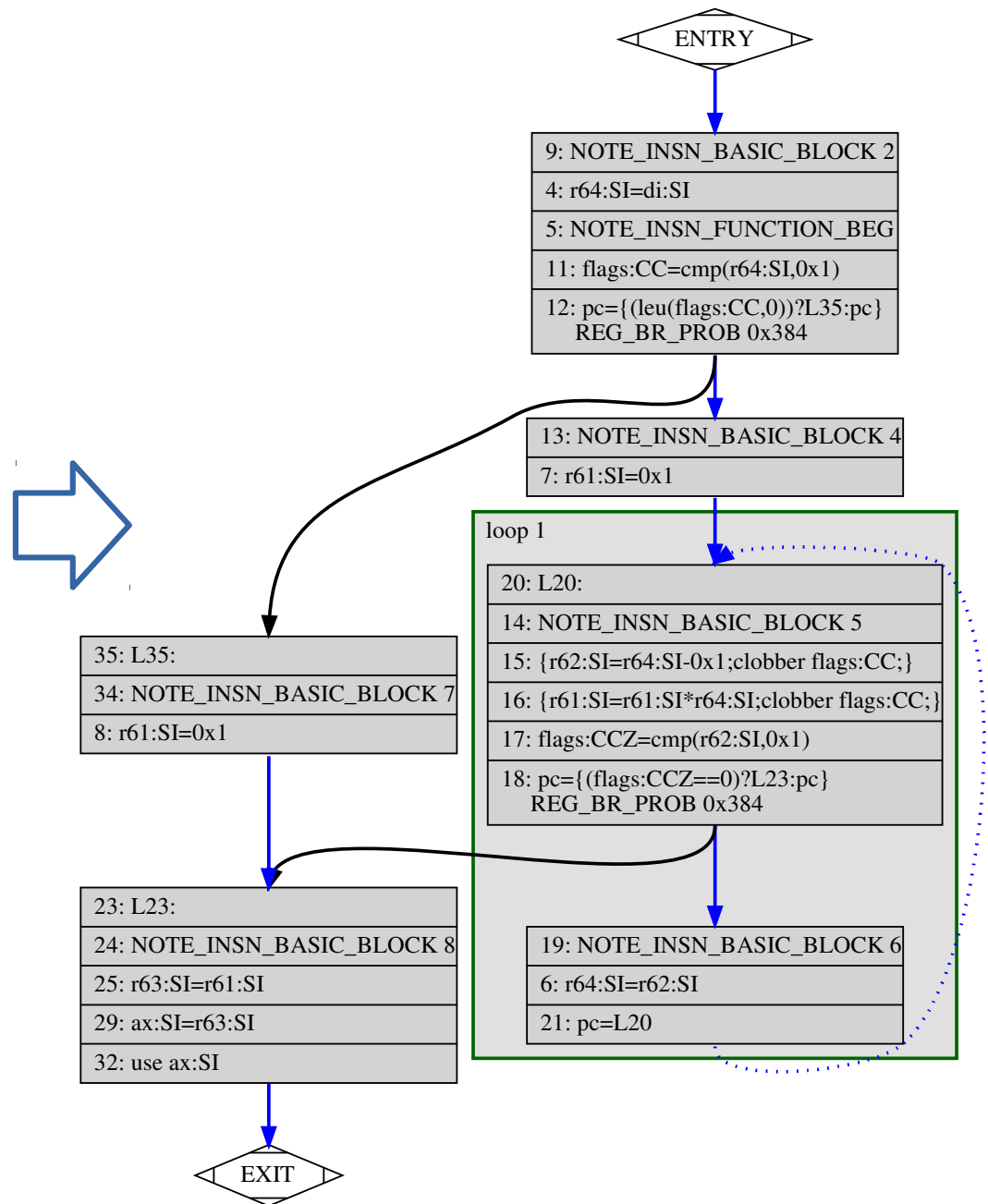


```
(insn 14 13 0 (parallel [  
  (set (reg:SI 89 [ _6 ])  
    (plus:SI (reg/v:SI 91 [ x ])  
      (const_int -1 [0xffffffffffffffff])))  
  (clobber (reg:CC 17 flags))  
) fact.c:7 -1  
(nil))
```

# RTL с виртуальными регистрами

```
unsigned
fact (unsigned x)
{
    if (x < 2)
        return 1;

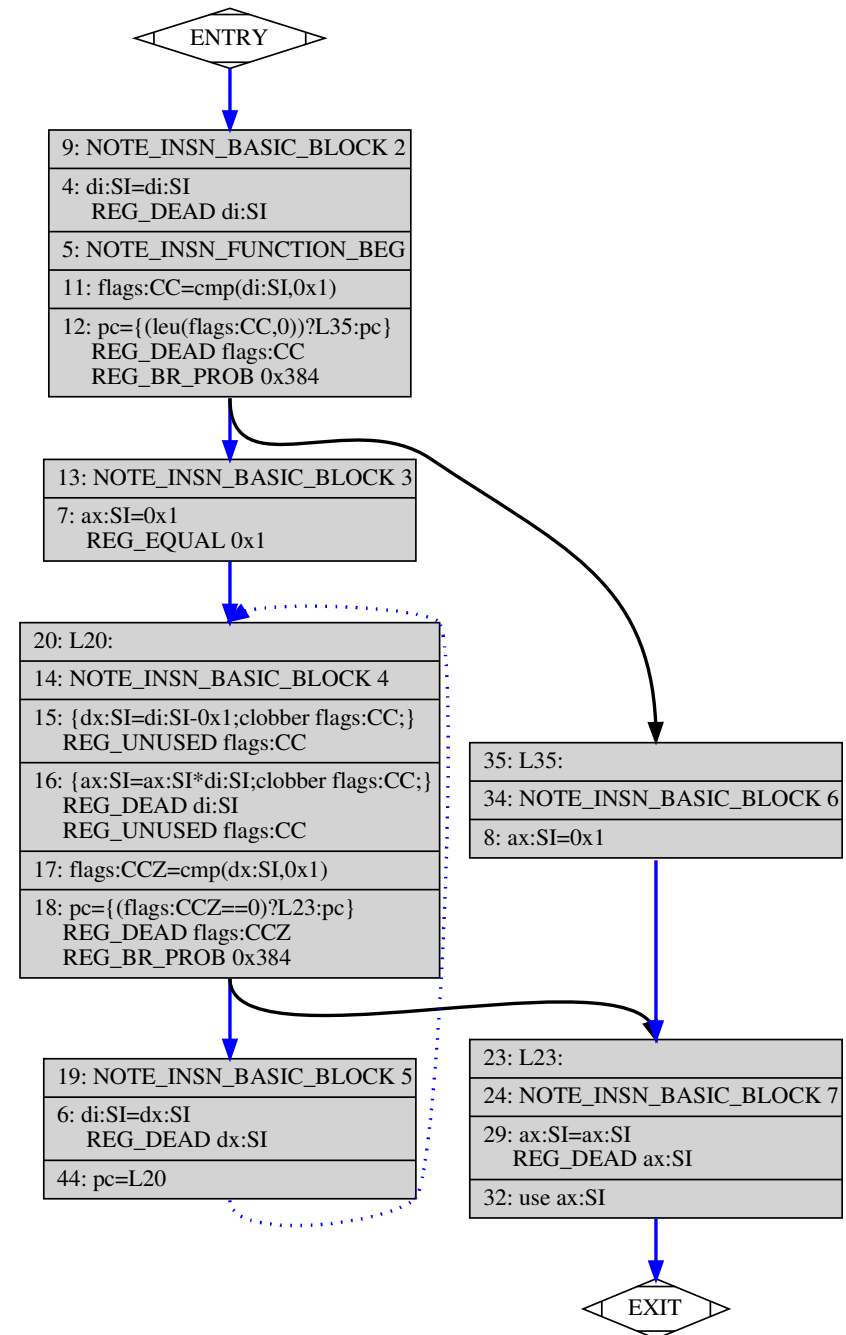
    return
        x * fact (x-1);
}
```



# RTL с физическими регистрами

```
unsigned
fact (unsigned x)
{
    if (x < 2)
        return 1;

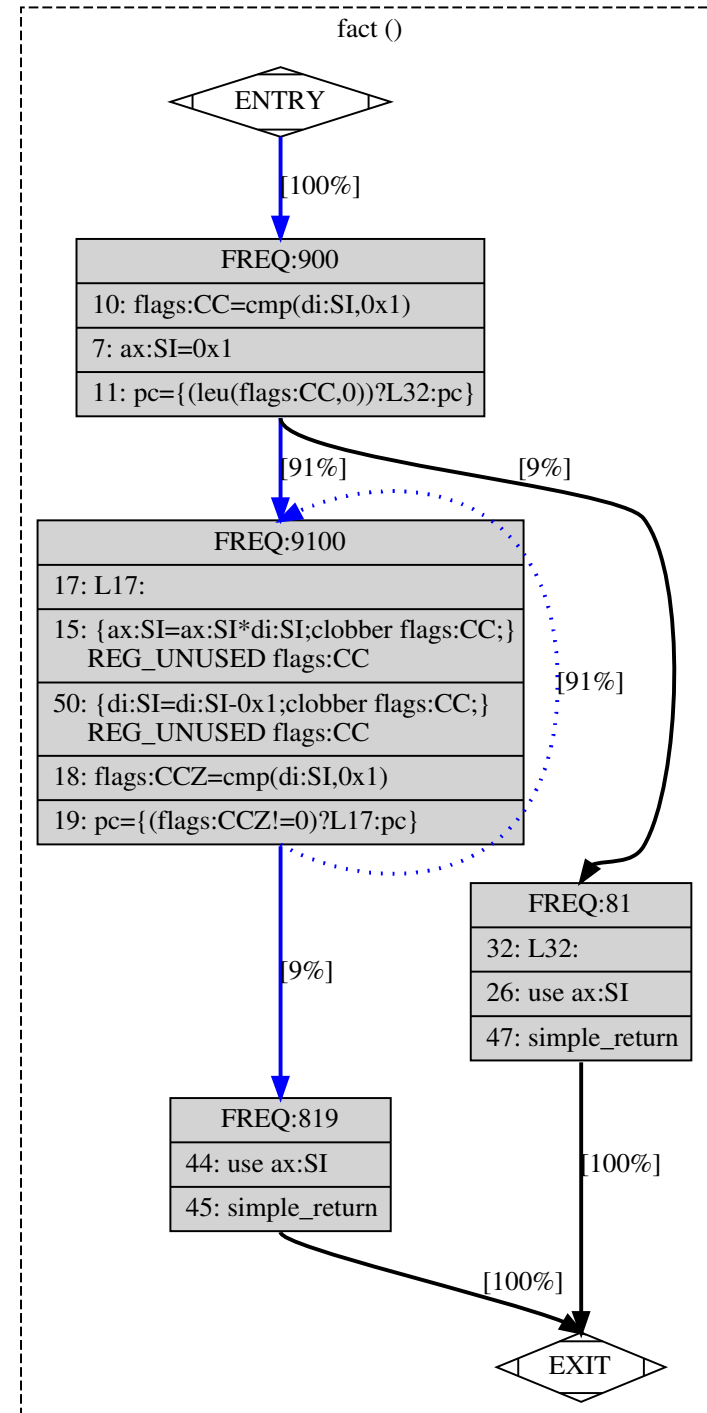
    return
        x * fact (x-1);
}
```



# Машинно-зависимые преобразования

```
unsigned
fact (unsigned x)
{
    if (x < 2)
        return 1;

    return
        x * fact (x-1);
}
```



# ИТОГОВЫЙ ассемблер

```
unsigned
fact (unsigned x)
{
    if (x < 2)
        return 1;

    return
        x * fact (x-1);
}
```



```
.p2align 4,,15
.globl fact
.type fact, @function
fact:
.LFB0:
    .cfi_startproc
    cmpl $1, %edi
    movl $1, %eax
    jbe .L4
    .p2align 4,,10
    .p2align 3
.L3:
    imull %edi, %eax
    subl $1, %edi
    cmpl $1, %edi
    jne .L3
    rep; ret
.L4:
    rep; ret
    .cfi_endproc
```

# Итоговый ассемблер: аннотация RTL

```
$ gcc -O2 -S fact.c -dP
```

```

#(insn 8 11 12 2 (set (reg:SI 0 ax [orig:61 D.1741 ] [61])
#      (const_int 1 [0x1])) fact.c:4 89 {*movsi_internal}
#      (nil))
      movl      $1, %eax      # 8      *movsi_internal/1  [length = 5]
#(jump_insn:TI 12 8 56 2 (set (pc)
#      (if_then_else (gtu (reg:CC 17 flags)
#      (const_int 0 [0]))
#      (label_ref:DI 20)
#      (pc))) fact.c:4 612 {*jcc_1}
#      (expr_list:REG_DEAD (reg:CC 17 flags)
#      (expr_list:REG_BR_PROB (const_int 9100 [0x238c])
#      (nil)))
# -> 20)
      ja      .L3      # 12      *jcc_1  [length = 2]
```



# Toolchain

- Системы компиляции
- Компилятор
- **Ассемблер**
- Линкер
- Отладка
- Производительность
- Портирование



# Ассемблер

- Язык ассемблера (например X86 ассемблер)
- Ассемблер как программа (например GAS)
- Ассемблирование как процесс

# Ассемблер X86: элементы языка

## Инструкции

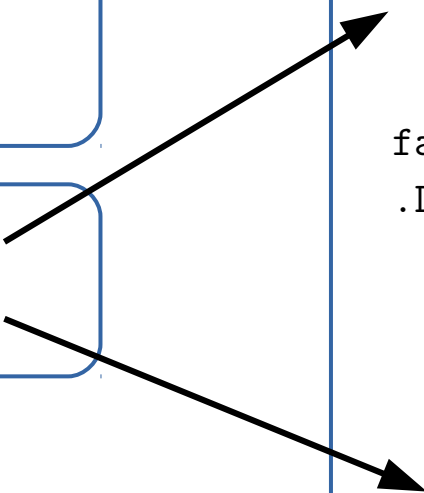


```
.text
.p2align 4,,15
.globl fact
.type fact, @function
fact:
.LFB0:
.cfi_startproc
    cmpl $1, %edi
    movl $1, %eax
    jbe .L4
    .p2align 4,,10
    .p2align 3
.L3:
    imull %edi, %eax
    subl $1, %edi
    cmpl $1, %edi
    jne .L3
    rep; ret
.L4:
    rep; ret
.cfi_endproc
```

# Ассемблер X86: элементы языка

Инструкции

Директивы



```
.text
.p2align 4,,15
.globl fact
.type fact, @function
fact:
.LFB0:
.cfi_startproc
    cmpl    $1, %edi
    movl    $1, %eax
    jbe     .L4
.p2align 4,,10
.p2align 3
.L3:
    imull   %edi, %eax
    subl    $1, %edi
    cmpl    $1, %edi
    jne     .L3
    rep; ret
.L4:
    rep; ret
.cfi_endproc
```

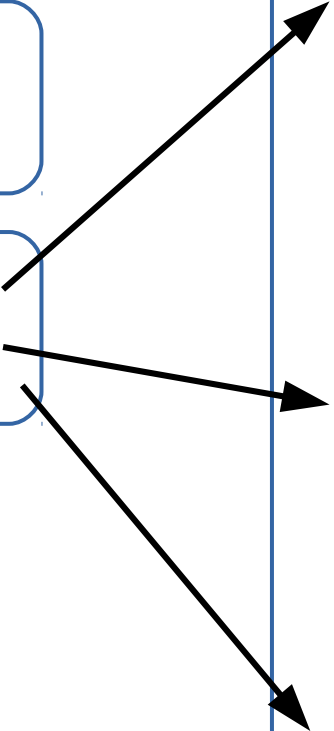
# Ассемблер X86: элементы языка

Инструкции

Директивы

Метки

```
.text
.p2align 4,,15
.globl fact
.type fact, @function
fact:
.LFB0:
.cfi_startproc
cmpl $1, %edi
movl $1, %eax
jbe .L4
.p2align 4,,10
.p2align 3
.L3:
imull %edi, %eax
subl $1, %edi
cmpl $1, %edi
jne .L3
rep; ret
.L4:
rep; ret
.cfi_endproc
```



# Ассемблер X86: элементы языка

Инструкции

Директивы

Метки

Секции

`.text`

`.p2align 4,,15`

`.globl fact`

`.type fact, @function`

`fact:`

`.LFB0:`

`.cfi_startproc`

`cmpl $1, %edi`

`movl $1, %eax`

`jbe .L4`

`.p2align 4,,10`

`.p2align 3`

`.L3:`

`imull %edi, %eax`

`subl $1, %edi`

`cmpl $1, %edi`

`jne .L3`

`rep; ret`

`.L4:`

`rep; ret`

`.cfi_endproc`

# Ассемблер X86: элементы языка

Инструкции

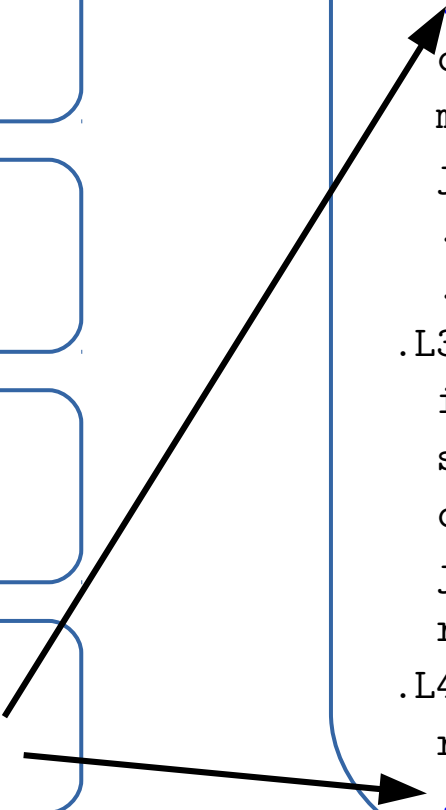
Директивы

Метки

Секции

DWARF

```
.text
.p2align 4,,15
.globl fact
.type fact, @function
fact:
.LFB0:
.cfi_startproc
    cmpl    $1, %edi
    movl    $1, %eax
    jbe     .L4
    .p2align 4,,10
    .p2align 3
.L3:
    imull   %edi, %eax
    subl    $1, %edi
    cmpl    $1, %edi
    jne     .L3
    rep; ret
.L4:
    rep; ret
.cfi_endproc
```



# Ассемблер X86: разный синтаксис

```
$ gcc -O2 -S -masm=[intel|att] program.c
```



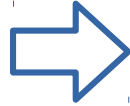
```
.globl fact
.type fact, @function
fact:
    cmp edi, 1
    mov eax, 1
    jbe .L4
    .p2align 4,,10
    .p2align 3
.L3:
    imul eax, edi
    sub edi, 1
    cmp edi, 1
    jne .L3
    rep ret
.L4:
    rep ret
```

```
.globl fact
.type fact, @function
fact:
    cmpl $1, %edi
    movl $1, %eax
    jbe .L4
    .p2align 4,,10
    .p2align 3
.L3:
    imull %edi, %eax
    subl $1, %edi
    cmpl $1, %edi
    jne .L3
    rep; ret
.L4:
    rep; ret
```



# Ассемблирование: машинный код

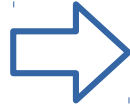
`cmpl $1, %edi`



`83 ff 01`

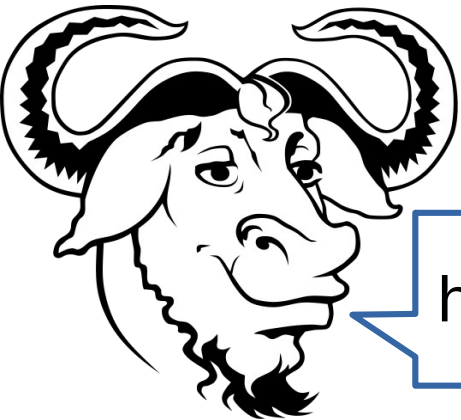
`SUB Ev, Ib`

`mov $0x1,%eax`



`b8 01 00 00 00`

`MOV EAX, Iv`



<http://sparksandflames.com/files/x86InstructionChart.html>

# Ассемблирование: сборка секций

.text  
..... some code .....  
.data  
..... some data .....  
.text  
..... more code .....  
.data  
..... more data .....  
.text  
..... more code .....  
.text  
..... more code .....

ELF object file header

.....

~~Program header table~~

.....

.text

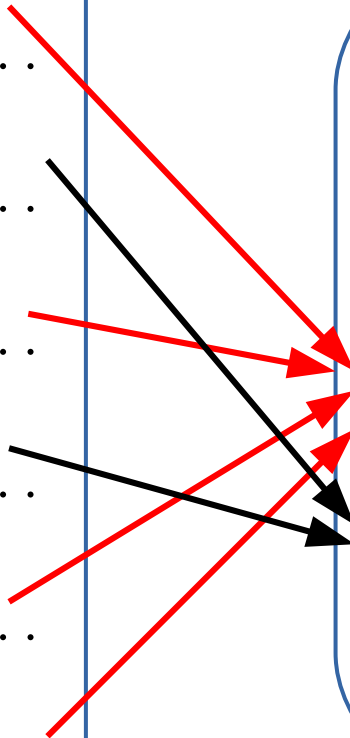
.... encoded instructions ...

.data

.....

Section header table

.....



# GAS: макроассемблер



# GAS: макроассемблер

- Макросы

```
.macro    sum from=0, to=5
    .long    \from
    .if      \to-\from
    sum      "(\from+1)",\to
    .endif
.endm
```

Как использовать этот макрос?

# GAS: макроассемблер

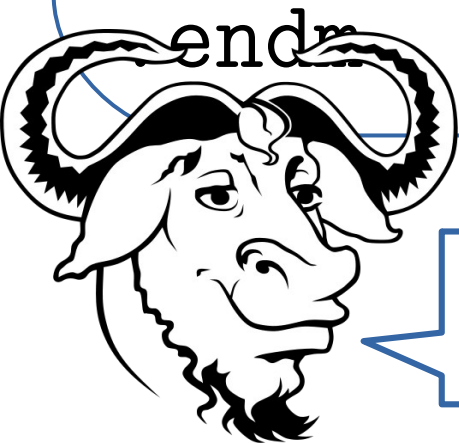
- Макросы

```
.macro    sum from=0, to=5  
    .long    \from  
    .if      \to-\from  
    sum      "(\from+1)", \to  
    .endif
```

sum 0, 5



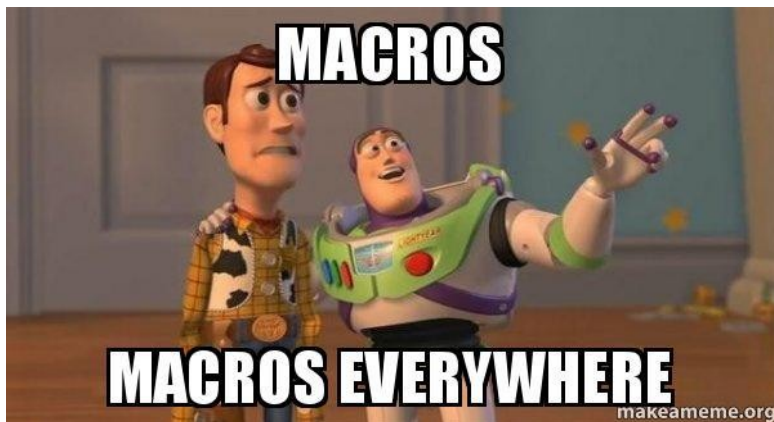
```
.long 0  
.long 1  
...  
.long 5
```



<https://sourceware.org/binutils/docs/as/Macro.html>

# GAS: макроассемблер

- Макросы



- Включение файлов

```
.macro    sum from=0, to=5  
    .long    \from  
    .if      \to-\from  
    sum      "(\from+1)", \to  
    .endif  
.endm
```

```
.include "other.s"
```

# GAS: макроассемблер

- Макросы



- Включение файлов
- Комментарии

```
.macro  sum from=0, to=5
    .long    \from
    .if      \to-\from
    sum      "(\from+1)", \to
    .endif
.endm
```

```
.include "other.s"
```

```
ja     .L3    # 12    *jcc_1
```

# GAS: макроассемблер

- Макросы



- Включение файлов
- Комментарии
- Дополнительные возможности

```
.macro    sum from=0, to=5  
    .long    \from  
    .if      \to-\from  
    sum      "(\from+1)", \to  
    .endif  
.endm
```

```
.include "other.s"
```

```
ja      .L3      # 12      *jcc_1
```

```
melvin: .long .
```

Что содержит melvin?



# GAS: макроассемблер

- Макросы



- Включение файлов
- Комментарии
- Дополнительные возможности

```
.macro    sum from=0, to=5
    .long    \from
    .if      \to-\from
    sum      "(\from+1)", \to
    .endif
.endm
```

```
.include other.s
```

```
ja      .L3      # 12      *jcc_1
```

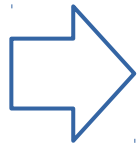
```
// melvin = &melvin
melvin: .long .
```

# Локальные метки

```
.L1:
    cmpl    $2, %edi
    jg      .L2
    cmpl    $-9, %edi
    jge     .L1

.L2:
    cmpl    $-9, %edi
    jge     .L3
    cmpl    $2, %edi
    jle     .L3
    cmpl    $0, %edi
    jg      .L2

.L3:
```



```
1:
    cmpl    $2, %edi
    jg      1f
    cmpl    $-9, %edi
    jge     1b

1:
    cmpl    $-9, %edi
    jge     1f
    cmpl    $2, %edi
    jle     1f
    cmpl    $0, %edi
    jg      1b

1:
```

# objdump: дизассемблирование

```
$ as fact.s --64 -o fact.o
```

```
$ objdump -d fact.o > fact.d
```

```
fact:
    cmpl    $1, %edi
    movl    $1, %eax
    jbe     .L4
    .p2align 4,,10
    .p2align 3
.L3:
    imull   %edi, %eax
    subl    $1, %edi
    cmpl    $1, %edi
    jne     .L3
    rep; ret
.L4:
    rep; ret
```

```
fact.o:      file format elf64-x86-64
```

```
Disassembly of section .text:
```

```
0000000000000000 <fact>:
```

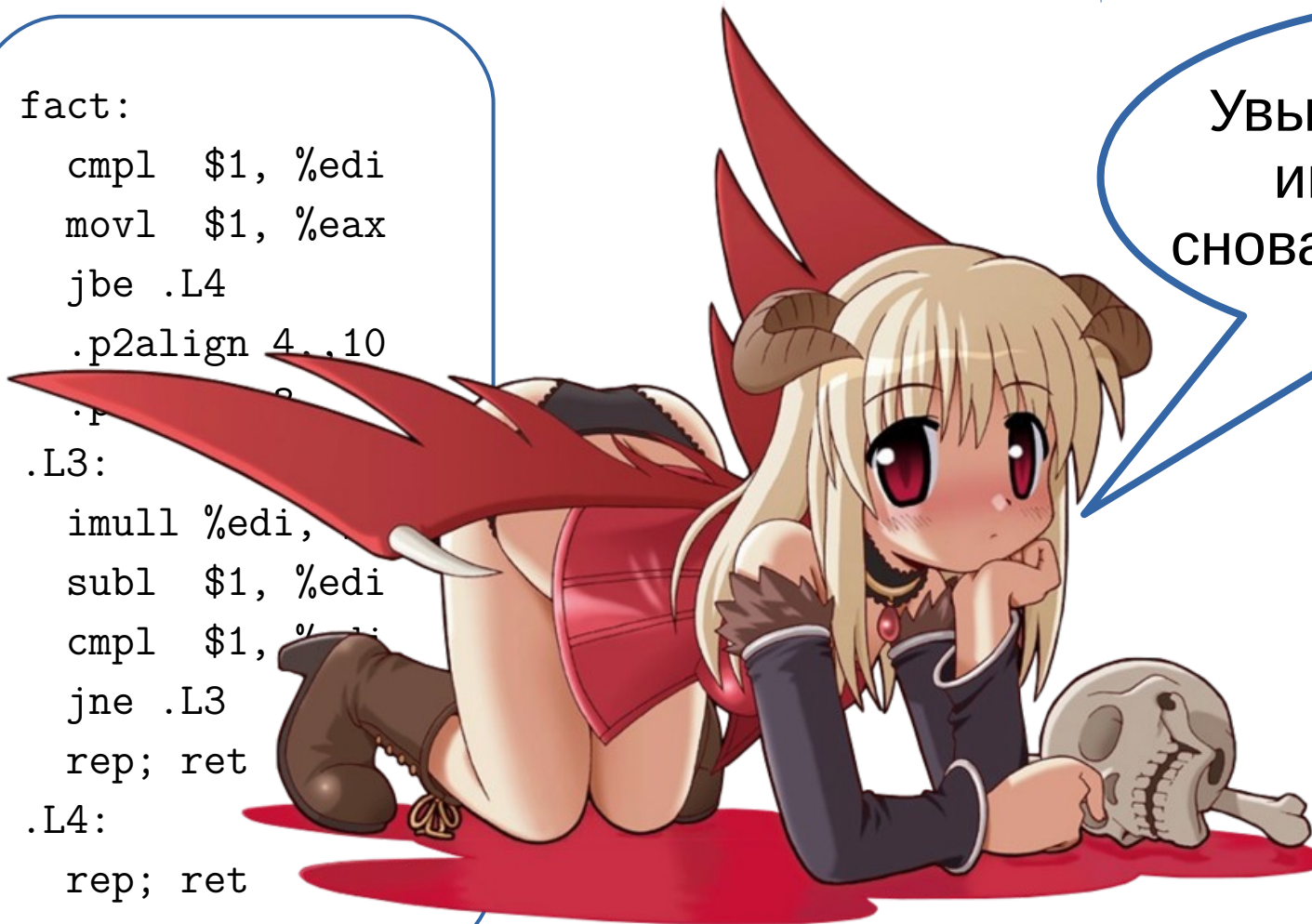
0:	83 ff 01	cmp	\$0x1,%edi
3:	b8 01 00 00 00	mov	\$0x1,%eax
8:	76 13	jbe	1d <fact+0x1d>
a:	66 0f 1f 44 00 00	nopw	0x0(%rax,%rax,1)
10:	0f af c7	imul	%edi,%eax
13:	83 ef 01	sub	\$0x1,%edi
16:	83 ff 01	cmp	\$0x1,%edi
19:	75 f5	jne	10 <fact+0x10>
1b:	f3 c3	repz retq	
1d:	f3 c3	repz retq	

# objdump: дизассемблирование

```
$ as fact.s --64 -o fact.o  
$ objdump -d fact.o > fact.d
```

```
fact:  
    cmpl    $1, %edi  
    movl    $1, %eax  
    jbe     .L4  
    .p2align 4,,10  
    .L3:  
    imull   %edi,  
    subl    $1, %edi  
    cmpl    $1, %edi  
    jne     .L3  
    rep; ret  
.L4:  
    rep; ret
```

Увы, дизассемблер  
иногда сложно  
снова ассемблировать



# Усложненный пример факториала

fact.c

```
unsigned
fact (unsigned x)
{
    if (x < 2)
        return 1;

    return
        x * fact (x-1);
}
```

main.c

```
int
main (void)
{
    unsigned f = fact(5);
    printf ("%u\n", f);
    return 0;
}
```

Как будут скомпилированы  
вызовы fact и printf в main.o?

# C : объявления и определения

- Объявление (declaration) вводит имя и сигнатуру для компилятора
- Определение (definition) содержит реализацию (тело функции, etc)

```
int printf (const char *fmt, ...)  
unsigned fact (unsigned);
```

```
int  
main (void) {  
    ..... call printf and fact .....  
}
```

libc.a

fact.o

# С : объявления и определения

- Объявление (declaration) вводит имя и сигнатуру для компилятора
- Определение (definition) содержит реализацию (тело функции, etc)

```
#include <stdio.h>
#include "fact.h"
```

```
int
main (void) {
..... call printf and fact .....
}
```

libc.a

fact.o

# Попытка собрать main.c

```
$ gcc -O2 main.c -save-temps  
main.o: In function `main':  
main.c:(.text+0xa): undefined reference to `fact'  
collect2: ld returned 1 exit status
```

```
$ readelf -s main.o  
9: 0000000000000000 43 FUNC      GLOBAL DEFAULT 1 main  
10: 0000000000000000 0 NOTYPE    GLOBAL DEFAULT UND fact  
11: 0000000000000000 0 NOTYPE    GLOBAL DEFAULT UND printf
```





# Попытка собрать main.c

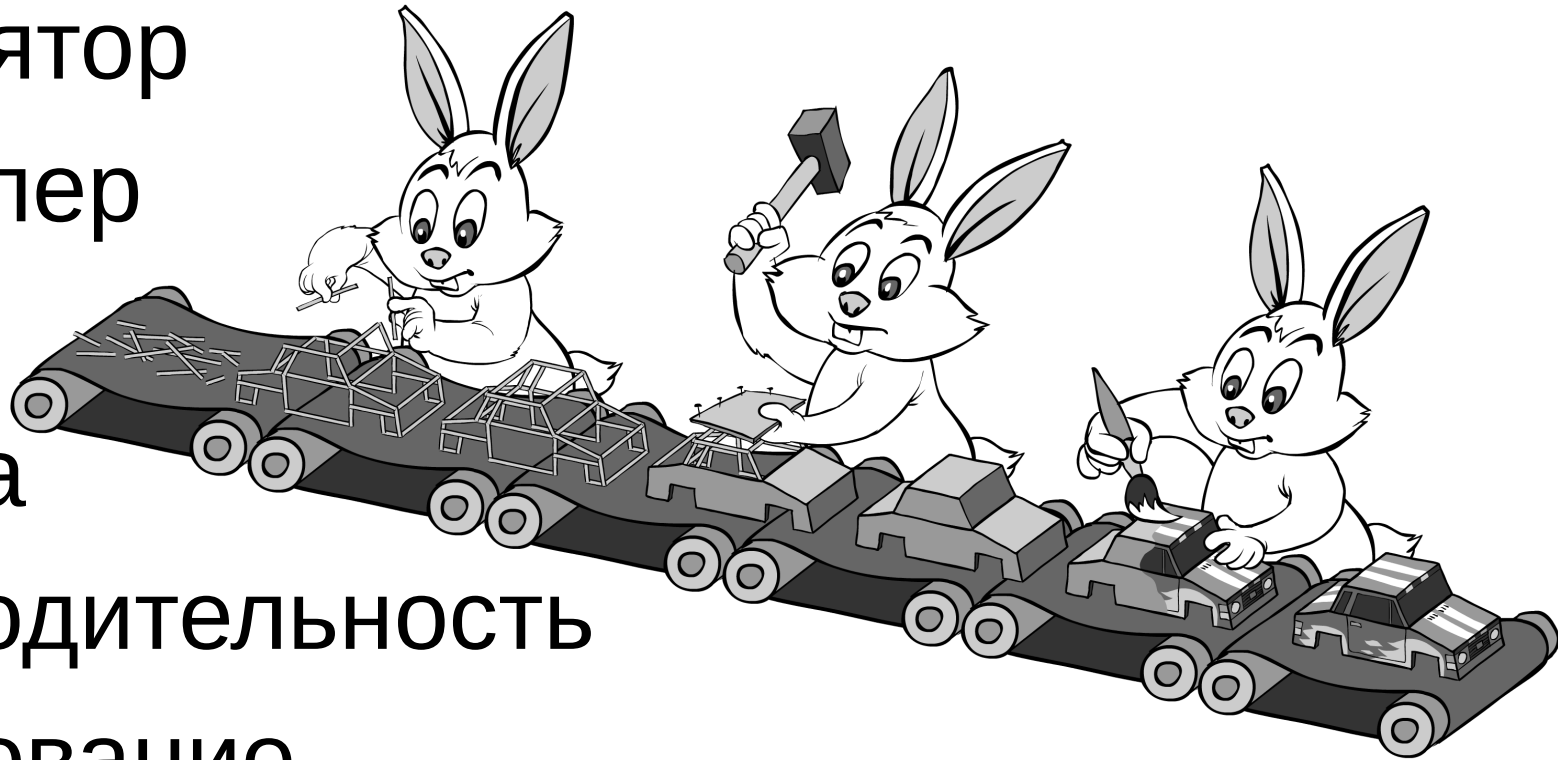
```
$ gcc -O2 main.c -save-temps  
main.o: In function `main':  
main.c:(.text+0xa): undefined reference to `fact'  
collect2: ld returned 1 exit status
```

```
$ readelf -s main.o  
9: 0000000000000000 43 FUNC GLOBAL DEFAULT 1 main  
10: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND fact  
11: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND printf
```

## Пора задействовать линкер!

# Toolchain

- Системы компиляции
- Компилятор
- Ассемблер
- **Линкер**
- Отладка
- Производительность
- Портирование



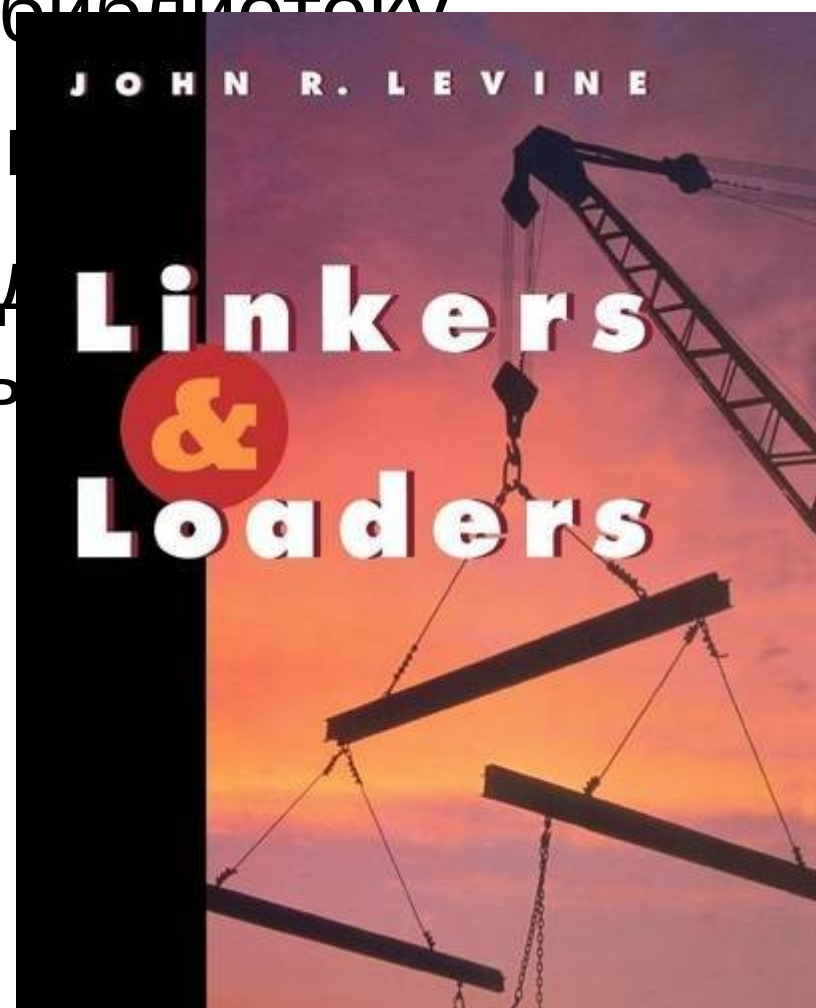
# Линковка (Linking)

- Сборка объектных модулей в исполняемый файл или динамическую библиотеку
- Разрешение межмодульных зависимостей
- Проверка ошибок неопределенных или многократно определенных символов



# Линковка (Linking)

- Сборка объектных модулей в исполняемый файл или динамическую библиотеку
- Разрешение межмодульных ссылок
- Проверка ошибок неопределенно определенных символов



# Статическая линковка

```
$ gcc fact.c main.c -static -save-temps
```

fact.c

```
unsigned
fact (unsigned x)
{
    if (x < 2)
        return 1;

    return
        x * fact (x-1);
}
```

main.c

```
int
main (void)
{
    unsigned f = fact(5);
    printf ("%u\n", f);
    return 0;
}
```

# Релокации

```
$ gcc fact.c main.c -static -save-temps  
$ objdump -d main.o > main.dis
```

000000000000000000 <main>:

0:	55	push	%rbp
1:	48 89 e5	mov	%rsp,%rbp
4:	bf 05 00 00 00	mov	\$0x5,%edi
9:	<b>e8</b> 00 00 00 00	callq	e <main+0xe>
e:	89 c2	mov	%eax,%edx

....

# Релокации

```
$ gcc fact.c main.c -static -save-temps  
$ objdump -d a.out > result.dis
```

0000000000400434 <main>:

400434:	55	push	%rbp
400435:	48 89 e5	mov	%rsp,%rbp
400438:	bf 05 00 00 00	mov	\$0x5,%edi
40043d:	<b>e8 1e 00 00 00</b>	callq	400460 <fact>
400442:	89 c2	mov	%eax,%edx

....

0000000000400460 <fact>:

400460:	55	push	%rbp
400461:	48 89 e5	mov	%rsp,%rbp

# Статическая линковка факториала: разбивка по шагам

```
$ gcc -O2 fact.c main.c -static -save-temps
```



```
$ gcc -O2 fact.c -c -o fact.o
```

```
$ gcc -O2 main.c -c -o main.o
```

```
$ gcc fact.o main.o -static -o fact.x
```



# Статическая линковка факториала: разбивка по шагам

```
$ gcc -O2 fact.c main.c -static -save-temps
```



```
$ gcc -O2 fact.c -c -o fact.o
```

```
$ gcc -O2 main.c -c -o main.o
```

```
$ ld fact.o main.o -static -o fact.x
```

# Статическая линковка факториала: разбивка по шагам

```
$ gcc -O2 fact.c main.c -static -save-temps
```



```
$ gcc -O2 fact.c -c -o fact.o
```

```
$ gcc -O2 main.c -c -o main.o
```

```
$ ld fact.o main.o -static -o fact.x
```

```
ld: warning: cannot find entry symbol _start;  
defaulting to 0000000000004000b0
```

```
main.o: In function `main':
```

```
main.c:(.text.startup+0x1d): undefined reference to  
`__printf_chk'
```

# Статические библиотеки

```
$ gcc fact.o main.o -static --verbose
```



```
collect2 -m elf_x86_64 -static crtbeginT.o -L.  
-L<other paths here> main.o fact.o --start-group  
-lgcc -lgcc_eh -lc --end-group crtend.o crtn.o
```

Где-то здесь прячется printf...



# Статические библиотеки своими руками

```
$ gcc -O2 fact.c -c -o fact.o  
$ ar -cr libfact.a fact.o  
$ gcc -static main.c -L. -lfact
```

- определение функции `fact` внутри `libfact.a`
- определение функции `printf` внутри `libc.a`
- общепринятое сокращение:  
`-lsmth == libsmth.a`
- упаковщик **ar** пакует объектные файлы в статические библиотеки

# Порядок линковки имеет значение

a.c

```
extern int a;  
int main() {  
    return a;  
}
```

b.c



libb.a

```
extern int b;  
int a;  
// ....  
a = b;
```

d.c



libd.a

```
int b;
```

```
$ gcc -L. -lb -ld a.c
```

```
$ gcc -L. -ld -lb a.c
```

```
$ gcc a.c -L. -ld -lb
```

```
$ gcc a.c -L. -lb -ld
```

Интересно,  
хоть один вариант  
правильный?



# Порядок линковки имеет значение

a.c

```
extern int a;  
int main() {  
    return a;  
}
```

b.c



libb.a

```
extern int b;  
int a;  
// ....  
a = b;
```

d.c



libd.a

```
int b;
```

\$ gcc -L. -lb -ld a.c



\$ gcc -L. -ld -lb a.c



\$ gcc a.c -L. -ld -lb



\$ gcc a.c -L. -lb -ld



# Порядок линковки имеет значение

a.c

```
extern int a;  
int main() {  
    return a;  
}
```

b.c



libb.a

```
extern int b;  
int a;  
// ....  
a = b;
```

d.c



libd.a

```
int b;
```

\$ gcc -L. -Wl,-\(-lb -ld -Wl,-\) a.c



\$ gcc -L. -Wl,-\(-ld -lb -Wl,-\) a.c



\$ gcc a.c -L. -Wl,-\(-ld -lb -Wl,-\)

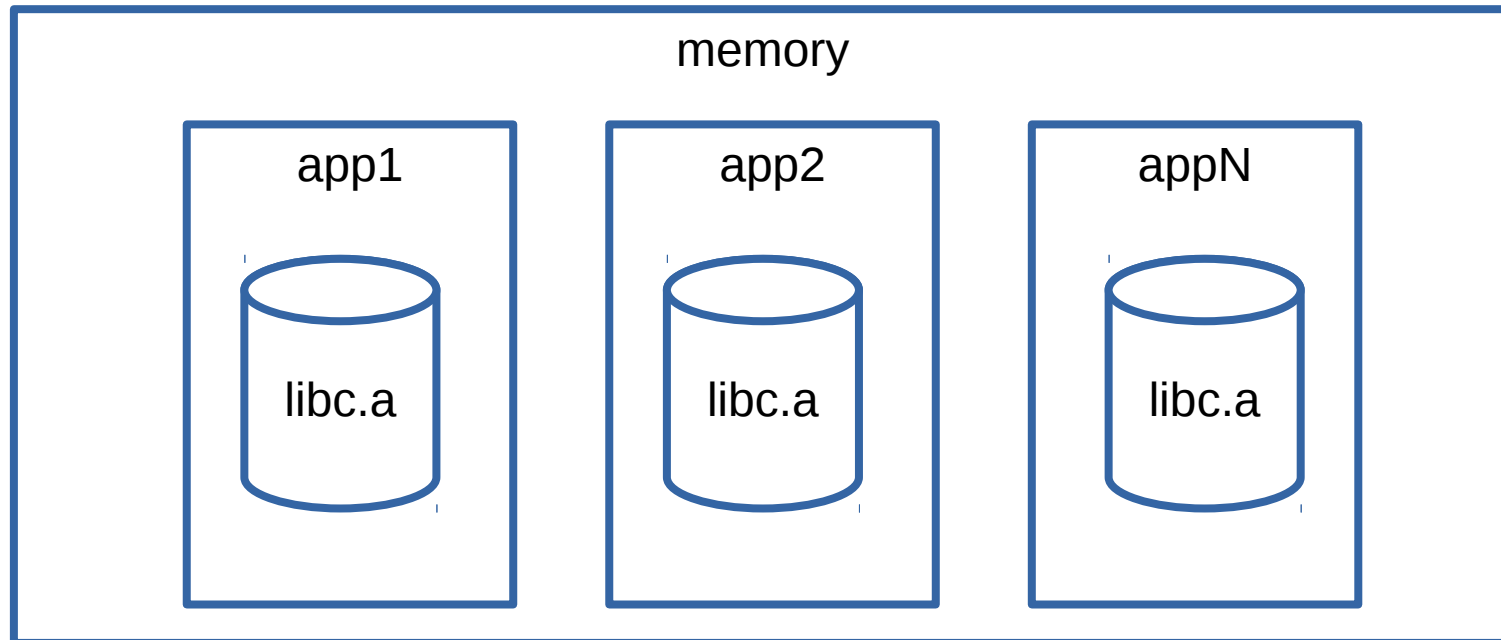


\$ gcc a.c -L. -Wl,-\(-lb -ld -Wl,-\)



# Проблемы статических библиотек

- Большой размер исполняемых файлов
- При исполнении нескольких файлов, библиотека оказывается загружена несколько раз





# Проблемы статических библиотек

- Большой размер исполняемых файлов
- При исполнении нескольких файлов, библиотека оказывается загружена несколько раз
- Решение: динамические библиотеки

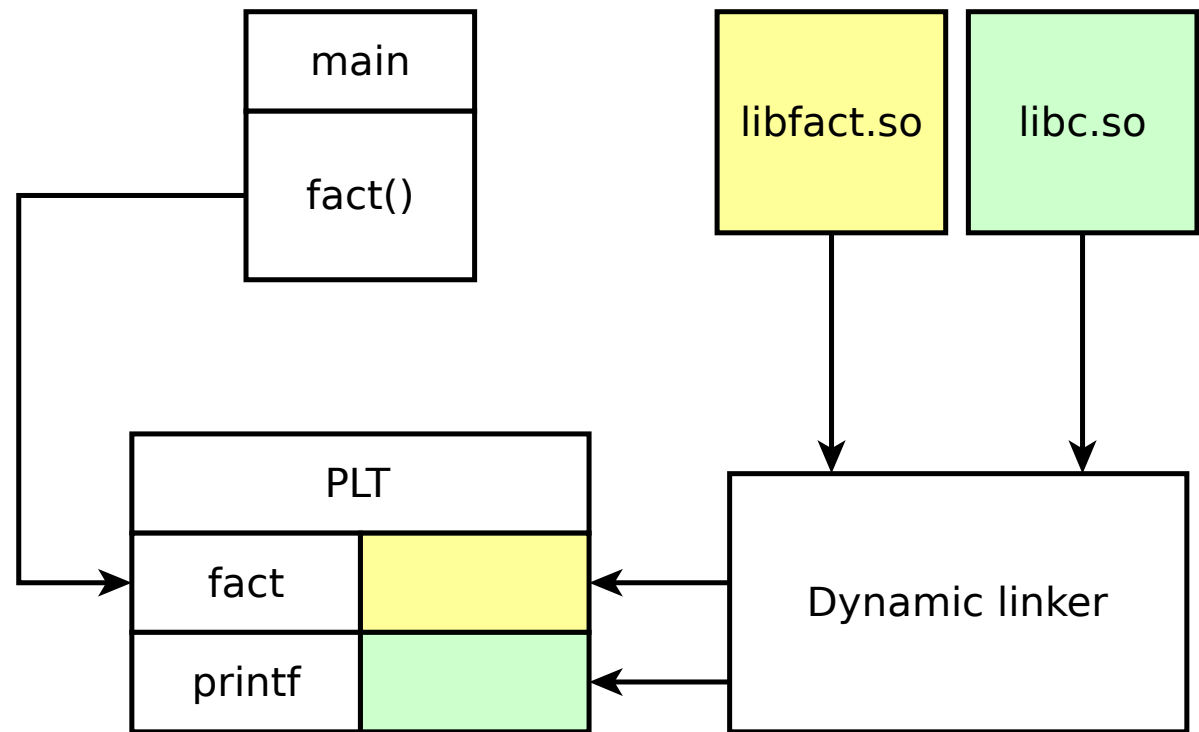
```
$ gcc -O2 -fPIC fact.c -c -o fact.o
```

```
$ gcc -shared fact.o -o libfact.so
```

```
$ gcc main.c -L. -lfact -Wl,-rpath,.
```

# Исследование динамической библиотеки: PLT

```
main:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
movl $5, %edi
call fact@PLT
movl %eax, %esi
leaq .LC0(%rip), %rdi
movl $0, %eax
call printf@PLT
```



# Исследование динамической библиотеки: GOT

```
unsigned lookup[] =  
{1, 1, 2, 6, 24, 120, 720,  
5040, 40320, 362880,  
3628800, 39916800,  
479001600};
```

```
unsigned fact (unsigned x)  
{  
    assert (x < 13);  
    return lookup[x];  
}
```

```
fact:  
.LFB0:  
.cfi_startproc  
cmpl    $12, %edi  
ja      .L5  
movl    %edi, %edi  
movl    lookup(,%rdi,4), %eax  
ret
```

```
$ gcc fact-lookup.c -S -O2
```

# Исследование динамической библиотеки: GOT

```
unsigned lookup[] =  
{1, 1, 2, 6, 24, 120, 720,  
5040, 40320, 362880,  
3628800, 39916800,  
479001600};
```

```
unsigned fact (unsigned x)  
{  
    assert (x < 13);  
    return lookup[x];  
}
```

```
fact:  
.LFB0:  
.cfi_startproc  
cmpl    $12, %edi  
ja      .L5  
movq  
    lookup@GOTPCREL(%rip), %rax  
movl    %edi, %edi  
movl    (%rax,%rdi,4), %eax  
ret
```

```
$ gcc fact-lookup.c -S -O2 -fPIC
```

# Исследование динамической библиотеки: GOT

```
extern
unsigned lookup[];

unsigned
fact (unsigned x)
{
    assert (x < 13);
    return lookup[x];
}
```



```
00000000004006dd <fact>:
4006dd:    push    %rbp
4006de:    mov     %rsp,%rbp
4006e1:    sub     $0x10,%rsp
4006e5:    mov     %edi,-0x4(%rbp)
4006e8:    cmpl    $0xc,-0x4(%rbp)
4006ec:    jbe     40070d <fact+0x30>
// some assert-related stuff
40070d:    mov     0x2008e4(%rip),%rax
400714:    mov     -0x4(%rbp),%edx
400717:    mov     (%rax,%rdx,4),%eax
40071a:    leaveq
40071b:    retq
```

```
$ gcc fact-lookup.c -L. -lfact-lookup-table
```

# Исследование динамической библиотеки: GOT

Надо найти  
GOTPCREL



```
00000000004006dd <fact>:
4006dd:    push    %rbp
4006de:    mov     %rsp,%rbp
4006e1:    sub     $0x10,%rsp
4006e5:    mov     %edi,-0x4(%rbp)
4006e8:    cmpl    $0xc,-0x4(%rbp)
4006ec:    jbe     40070d <fact+0x30>
// some assert-related stuff
40070d:    mov     0x2008e4(%rip),%rax
400714:    mov     -0x4(%rbp),%edx
400717:    mov     (%rax,%rdx,4),%eax
40071a:    leaveq
40071b:    retq
```

# Исследование динамической библиотеки: GOT

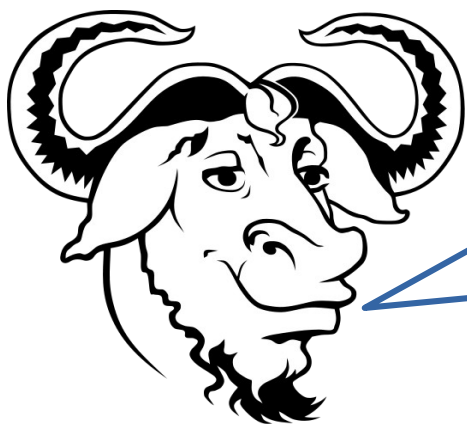
адрес 0x2008e4  
похож на  
глобальные данные



```
00000000004006dd <fact>:
4006dd:    push    %rbp
4006de:    mov     %rsp,%rbp
4006e1:    sub     $0x10,%rsp
4006e5:    mov     %edi,-0x4(%rbp)
4006e8:    cmpl    $0xc,-0x4(%rbp)
4006ec:    jbe     40070d <fact+0x30>
// some assert-related stuff
40070d:    mov     0x2008e4(%rip),%rax
400714:    mov     -0x4(%rbp),%edx
400717:    mov     (%rax,%rdx,4),%eax
40071a:    leaveq
40071b:    retq
```

# LDD: анализ динамических зависимостей и библиотек

```
$ ldd a.out
linux-vdso.so.1 => (0x00007fff6956b000)
libfact.so => ./libfact.so (0x00007f86adf83000)
libc.so.6 => /lib/libc.so.6 (0x00007f86adb2000)
/lib64/ld-linux-x86-64.so.2 (0x00007f86ae186000)
```



Пути здесь зависят от опции `rpath`,  
поданной на линковке

```
$ gcc -O2 main.c -L. -lfact -Wl,-rpath,.
```



# Реально подключенные библиотеки

```
$ gcc test.c -lfact -L. --verbose  
collect2 --eh-frame-hdr -m elf_x86_64  
-dynamic-linker ld-linux-x86-64.so.2 crt1.o crti.o  
crtbegin.o -L. -L/lib/../../lib64 -L/usr/lib/../../lib64  
test.o -lfact -lgcc -lc -lgcc --as-needed -lgcc_s  
--no-as-needed crtend.o crtn.o
```

- crtbegin, crtend, crt1, crti, crtn называются стартовыми файлами
- Именно там находятся настоящее начало и конец программы

# Проблемы динамических библиотек

- Большие накладные расходы на вызов функции (обращении к глобальным данным)
- Иногда хочется критичные библиотеки (lfact) линковать статически, а некритичные, но большие (lc) – динамически
- Решение: смешанная линковка

```
$ gcc test.c -Wl,-Bstatic -lfact -Wl,-Bdynamic -L.
```

- Опция -Wl транслирует опцию линкеру

# Проблемы динамических библиотек

- Слишком большое число экспортируемых функций увеличивает время загрузки

```
// user needs this
unsigned myfunc (unsigned);

// these ones can not be static, because library
// is multi-file, but they waste export table space
unsigned helper1 (unsigned);
unsigned helper2 (unsigned);
// .... here goes 40 more helpers
```

# Проблемы динамических библиотек

- Решение: управление видимостью

```
__attribute__((visibility(default)))  
unsigned myfunc (unsigned);  
  
unsigned helper1 (unsigned);  
unsigned helper2 (unsigned);  
// .... here goes 40 more helpers
```

```
$ gcc dl1.c dl2.c -fvisibility=hidden
```



Больше информации здесь: <https://gcc.gnu.org/wiki/Visibility>

# Подмена динамических библиотек

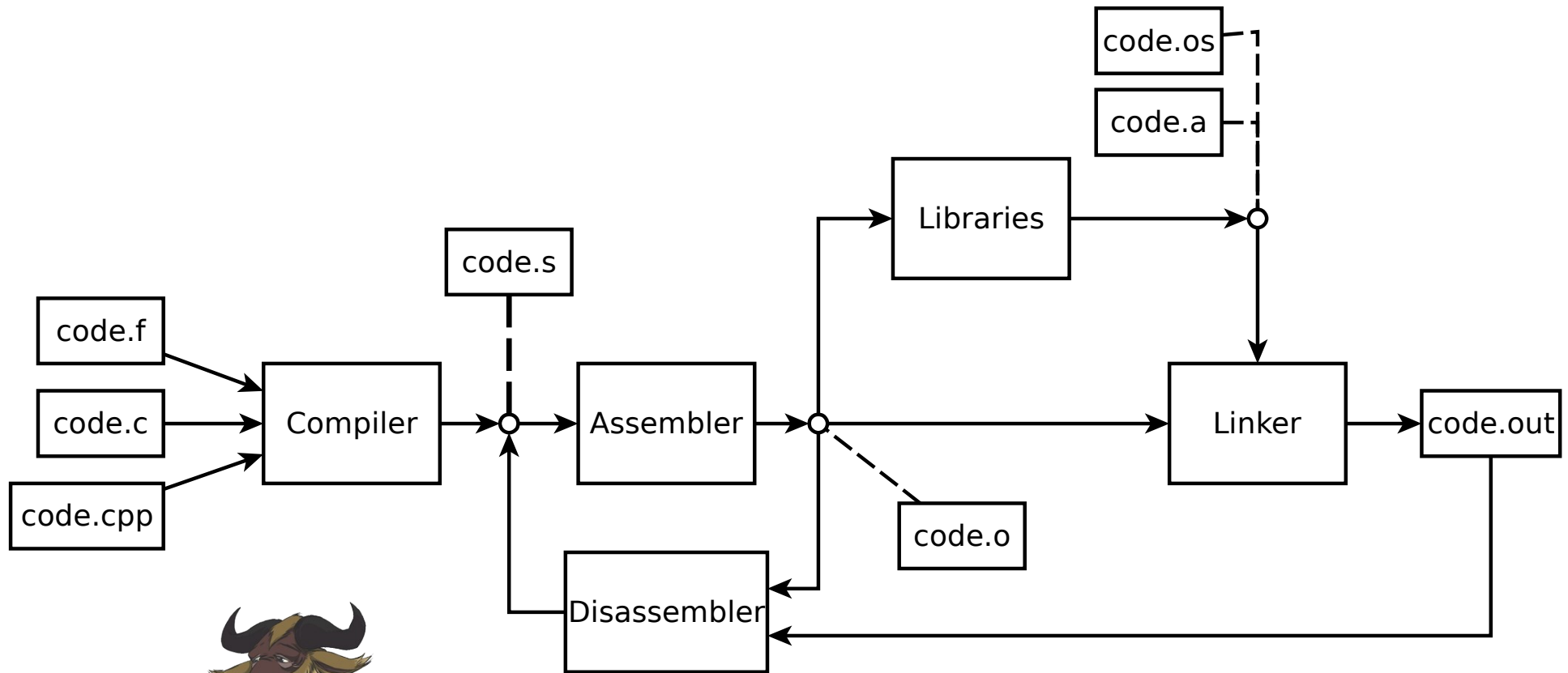
- Динамический линкер заполняет PLT/GOT на каждом запуске программы
- Что если те же таблицы заполнить до него чем-то другим?
- Для подмены используется LD\_PRELOAD

```
$ gcc -dynamic -O2 myprog.c -o a.out
```

```
$ LD_PRELOAD=/path/to/some.so a.out
```

- Эта немного хакерская техника очень полезна (пример: failmalloc)

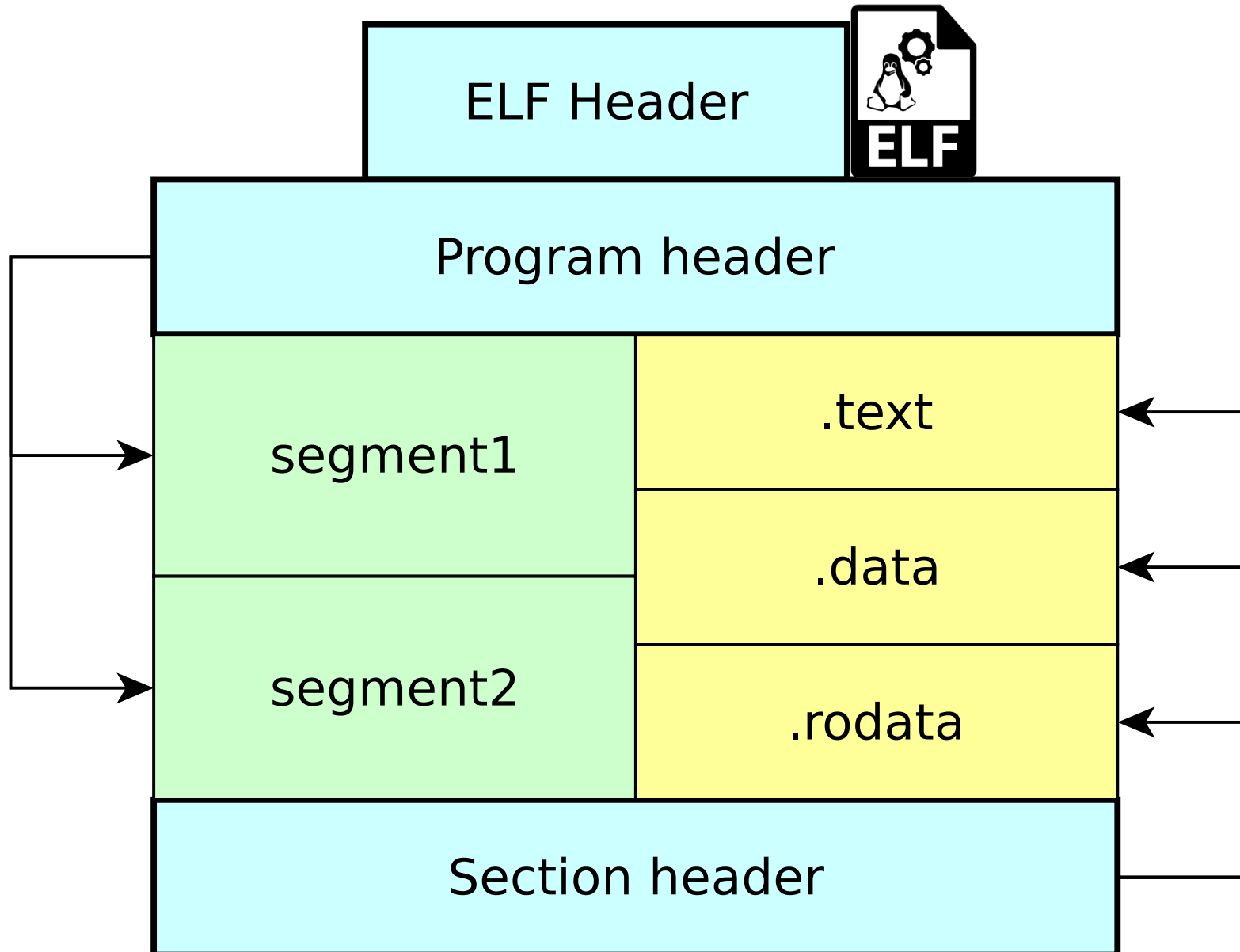
# Уточнённая схема toolchain



# Форматы выходных файлов

- ELF – де-факто стандарт для исполняемого и объектного кода в Unix-подобных системах
- PE – де-факто стандарт (включая свою доработку PE32+) для исполняемого кода в Windows-подобных системах
- COFF – популярен (как и его доработка ECOFF) для объектных файлов
- a.out – древний, но всё ещё популярный за свою минималистичность формат

# ELF – executable and linkable format





# Исследование section header

```
$ readelf -S a.out --wide
```

[Nr]	Name	Type	Address	Off	Size	ES	Flg	Lk	Inf	Al
[ 0]		NULL	0000000000000000	000000	000000	00		0	0	0
[ 1]	.interp	PROGBITS	0000000000400238	000238	00001c	00	A	0	0	1
[ 2]	.note.ABI-tag	NOTE	0000000000400254	000254	000020	00	A	0	0	4
[ 3]	.note.gnu.build-id	NOTE	0000000000400274	000274	000024	00	A	0	0	4
[ 4]	.gnu.hash	GNU_HASH	0000000000400298	000298	00001c	00	A	5	0	8
[ 5]	.dynsym	DYNSYM	00000000004002b8	0002b8	000060	18	A	6	1	8
[ 6]	.dynstr	STRTAB	0000000000400318	000318	00003f	00	A	0	0	1
[ 7]	.gnu.version	VERSYM	0000000000400358	000358	000008	02	A	5	0	2
[ 8]	.gnu.version_r	VERNEED	0000000000400360	000360	000020	00	A	6	1	8
[ 9]	.rela.dyn	RELA	0000000000400380	000380	000018	18	A	5	0	8
[10]	.rela.plt	RELA	0000000000400398	000398	000048	18	A	5	12	8
[11]	.init	PROGBITS	00000000004003e0	0003e0	00001a	00	AX	0	0	4
[12]	.plt	PROGBITS	0000000000400400	000400	000040	10	AX	0	0	16
[13]	.text	PROGBITS	0000000000400440	000440	0001b2	00	AX	0	0	16
[14]	.fini	PROGBITS	00000000004005f4	0005f4	000009	00	AX	0	0	4
[15]	.rodata	PROGBITS	0000000000400600	000600	00000d	00	A	0	0	4
[16]	.eh_frame_hdr	PROGBITS	0000000000400610	000610	00003c	00	A	0	0	4
.... и так далее ....										

# Объединения секций

- Хранение констант в секции `.rodata` создаёт некоторое дублирование после линковки. Ниже показаны две одинаковые константы в одном итоговом файле
- Хорошо бы объединять одинаковые секции

```
// foo.c
int foo() {
    float x = 1.0f
    ..... etc .....
// bar.c
int bar() {
    float x = 1.0f
    ..... etc .....
```



```
// foo.s
.section .rodata
.LC0:
.long 1065353216
// bar.s
.section .rodata
.LC0:
.long 1065353216
```



```
<foo>:
....
mov LB0,%eax
....
<bar>:
....
mov LB1,%eax
....
```

# Объединения секций

- gcc на O2 может помещать константы в специальные секции `rodata.cst.<size>` где они потом обрабатываются и объединяются линкером
- Это отключается через `-fno-merge-constants`

```
// foo.c
int foo() {
    float x = 1.0f
    ..... etc .....
// bar.c
int bar() {
    float x = 1.0f
    ..... etc .....
```



```
// foo.s
.section .rodata.cst.4
.LC0:
.long 1065353216
// bar.s
.section .rodata.cst.4
.LC0:
.long 1065353216
```



```
<foo>:
....
mov LB0,%eax
....
<bar>:
....
mov LB0,%eax
....
```

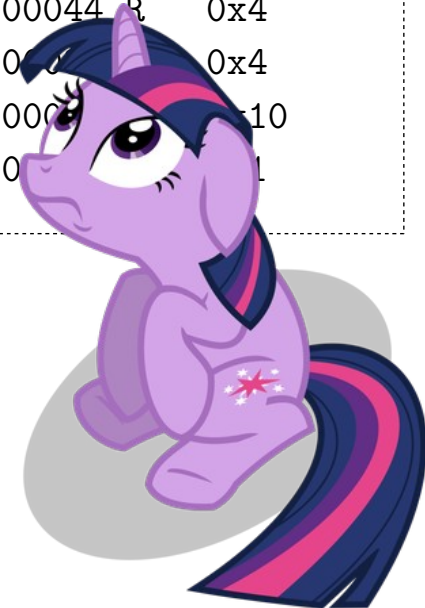
# Исследование program header

```
$ readelf -l a.out --wide
```

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000040	0x00000000000400040	0x00000000000400040	0x0001f8	0x0001f8	R E	0x8
INTERP	0x000238	0x00000000000400238	0x00000000000400238	0x00001c	0x00001c	R	0x1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]							
LOAD	0x000000	0x00000000000400000	0x00000000000400000	0x000764	0x000764	R E	0x200000
LOAD	0x000e10	0x00000000000600e10	0x00000000000600e10	0x000230	0x000238	RW	0x200000
DYNAMIC	0x000e28	0x00000000000600e28	0x00000000000600e28	0x0001d0	0x0001d0	RW	0x8
NOTE	0x000254	0x00000000000400254	0x00000000000400254	0x000044	0x000044	R	0x4
GNU_EH_FRAME	0x000610	0x00000000000400610	0x00000000000400610	0x00003c	0x00003c		0x4
GNU_STACK	0x000000	0x00000000000000000	0x00000000000000000	0x000000	0x000000		0x10
GNU_RELRO	0x000e10	0x00000000000600e10	0x00000000000600e10	0x0001f0	0x0001f0		0x1

Интересно, а как они соотносятся  
с секциями? Их так много...



# Исследование program header

```
$ readelf -l a.out --wide
```

Section to Segment mapping:

Segment Sections...

```
00
01      .interp
02      .interp .note.ABI-tag .note.gnu.build-id .gnu.hash
.dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt
.init .plt .text .fini .rodata .eh_frame_hdr .eh_frame
03      .init_array .fini_array .jcr .dynamic .got .got.plt .data
.bss
04      .dynamic
05      .note.ABI-tag .note.gnu.build-id
06      .eh_frame_hdr
07
08      .init_array .fini_array .jcr .dynamic .got
```

# Gold linker

- Проблемы стандартного GNU LD
  - однопоточность
  - медленная работа с большими объёмами кода
- Решение (только для ELF): gold
- В своё время был разработан как внутренний проект Google, сейчас официально часть GNU Binutils



```
$ gcc main.c fact.c -fuse-ld=gold
```

# Проблема межмодульной оптимизации

anynum.c

```
int  
any_number() {  
    return 5;  
}
```

main.c

```
int  
main() {  
    return any_number();  
}
```

```
$ gcc main.c anynum.c -O2
```

- Будет ли эта программа сооптимизирована путём инлайн-подстановки?

# Проблема межмодульной оптимизации

anynum.c

```
int  
any_number() {  
    return 5;  
}
```

main.c

```
int  
main() {  
    return any_number();  
}
```

```
$ gcc main.c anynum.c -O2
```

- Нет, не будет
- Основная проблема: separate translation



# Решение: LTO

```
$ gcc main.c anynum.c -O2 -flto
```

- LTO (от **link-time** optimizations)
- Аннотирует объектные файлы исходным кодом
- Использует плагин линкера для оптимизаций аннотированных объектных файлов во **время линковки**

# Скрипты линкера

```
$ gcc -Wl,--verbose
```

```
using internal linker script:
```

```
=====
```

```
/* Script for -z combreloc: combine and sort reloc sections */
```

```
OUTPUT_FORMAT("elf64-x86-64", "elf64-x86-64", "elf64-x86-64")
```

```
OUTPUT_ARCH(i386:x86-64)
```

```
ENTRY(_start)
```

```
SEARCH_DIR("/usr/local/lib64");
```

```
SECTIONS
```

```
{
```

```
    PROVIDE (__executable_start = SEGMENT_START("text-segment",  
0x400000));
```

```
    . = SEGMENT_START("text-segment", 0x400000) + SIZEOF_HEADERS;
```

```
    .interp          : { *(.interp) }
```

```
    .note.gnu.build-id : { *(.note.gnu.build-id) }
```

```
.... и так далее ....
```

# Скрипты линкера своими руками

```
SECTIONS { ❶  
    . = 0x00000000; ❷  
    .text : { ❸  
        abc.o (.text);  
        def.o (.text);  
    } ❹  
}
```

Директива  
SECTIONS

Счетчик  
локаций

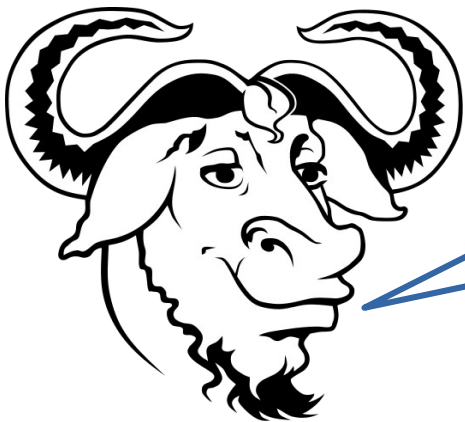
Начало и конец  
секций

```
SECTIONS {  
    . = 0x00000000;  
    .text : { * (.text); }  
}
```

Матчим все  
объектные файлы

# Полезные директивы

- ENTRY – определяет входную точку программы (по умолчанию `__start`)
- OUTPUT\_FORMAT – формат выходного файла (по умолчанию ELF)
- STARTUP – файл (`crt0`, `crti`) который должен быть прилинкован в начале исполняемого



Больше полезных директив на  
[sourceware.org/binutils/docs/ld/Scripts.html](http://sourceware.org/binutils/docs/ld/Scripts.html)

# Искусство objcopy

```
$ dd if=/dev/urandom of=blob.bin bs=1 count=16
$ objcopy -I binary -O elf64-x86-64 -B i386 blob.bin blob.o
$ objdump -t blob.o

blob.o:          file format elf64-x86-64

SYMBOL TABLE:
00000000 1      d  .data 00000000 .data
00000000 g          .data 00000000 _binary_blob_bin_start
00000010 g          .data 00000000 _binary_blob_bin_end
00000010 g          *ABS* 00000000 _binary_blob_bin_size
```

```
extern unsigned char _binary_blob_bin_start;
extern unsigned char _binary_blob_bin_end;
extern unsigned char _binary_blob_bin_size;
/* .... */
unsigned char *pblob = &_amp;_binary_blob_bin_start;
```

# ИскYCCTBO addr2line

```
1. int main()  
2. {  
3.     int *p = (int*)0xDEADBEEF;  
4.     *p = 5; /* boom */  
5. }
```

Segmentation fault

```
$ dmesg | tail -1
```

```
[ 4051.415509] a.out[3346]: segfault at deadbeef
```

```
ip 0x04004da sp 0x07ffdd0a97c0 error 6 in a.out[400000+1000]
```

```
$ gcc -g segf.c
```

```
$ addr2line -e a.out 0x04004da
```

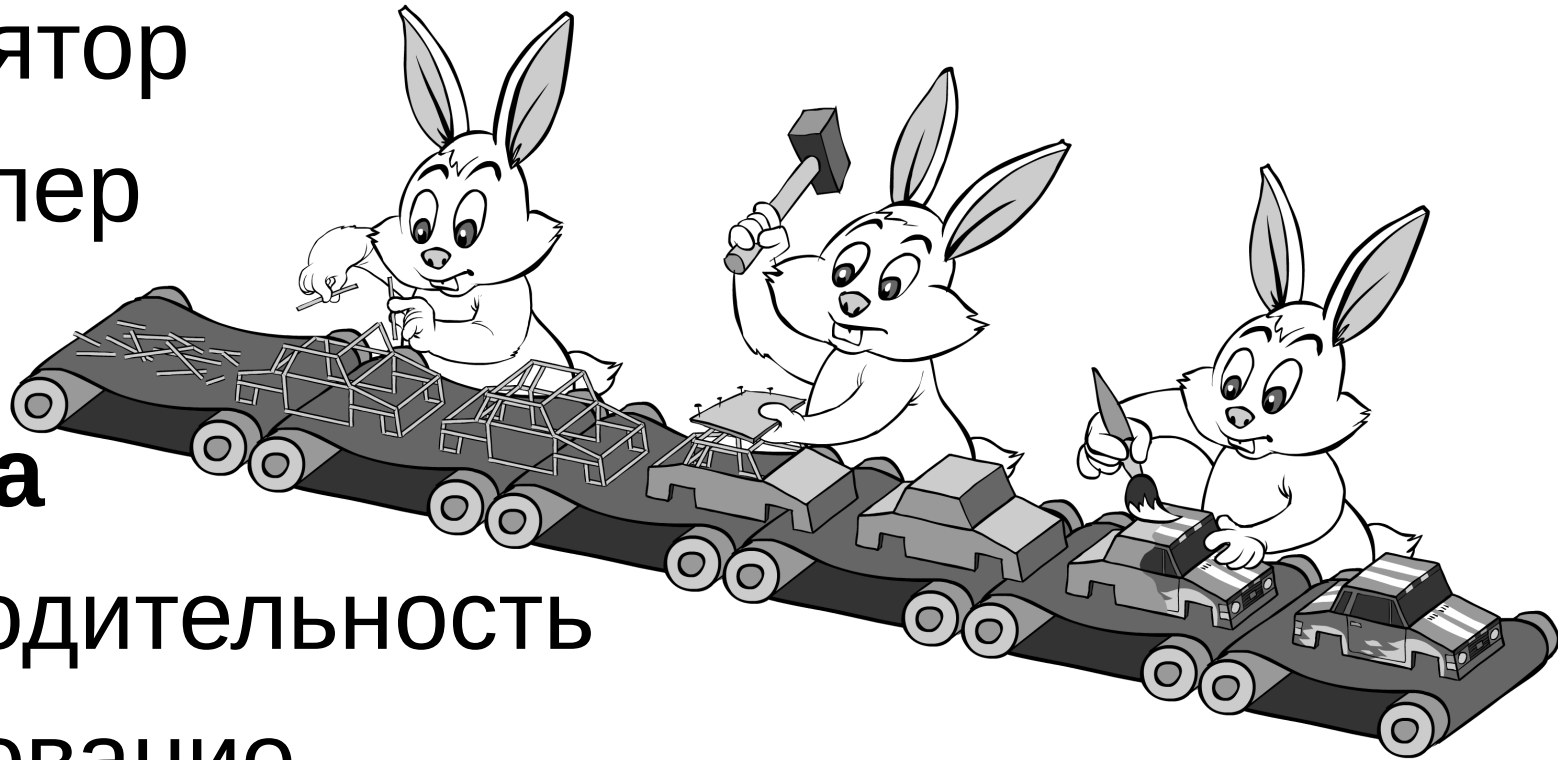
```
segf.c:4
```

# Стандартные библиотеки

- Glibc (GNU C Library)
  - Microsoft C Run-time Library
  - Dietlibc
  - uClibc
  - Newlib
  - Klibc
- Легковесные реализации  
для embedded систем
- Bionic – реализация стандартной библиотеки в Android

# Toolchain

- Системы компиляции
- Компилятор
- Ассемблер
- Линкер
- **Отладка**
- Производительность
- Портирование





# Отладчики (Debuggers)

- Отладчики микроархитектуры, такие как OpenOCD (on-chip debugger)
- Отладчики уровня ядра, такие как KGDB или WinDbg
- Отладчики пользовательского уровня, такие как GDB, LLDB, IDB, и прочие



**GDB**  
The GNU Project  
Debugger

# Работа с GDB

```
/* Simple but buggy bubble sort */
void
sort(item *a, int n) {
    int s, i = 0, j = 0;
    for (; i < n && s != 0; i++) {
        s = 0;
        for (j = 0; j < n; j++)
            if (a[j].key > a[j+1].key) {
                item t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
                s++;
            }
        n--;
    }
}
```

Настало время  
отладки...



# Работа с GDB

```
$ gcc buggy-sort.c -O0 -g -o buggy
```

```
$ gdb buggy
```

```
(gdb) break sort
```

```
Breakpoint 1 at 0x40056f: file buggy-sort.c, line 23.
```

```
(gdb) run
```

```
Breakpoint 1, sort (a=0x601040, n=5) at buggy-sort.c:23
```

```
23    int i = 0, j = 0;
```

```
(gdb) next
```

```
26    for (; i < n && s != 0; i++)
```

```
(gdb) print s
```

```
$1 = 0
```

# Отладчик LLDB

- Основная проблема GDB: там отдельный парсер для C++

```
$ gdb a.out  
gdb> start  
gdb> p [y = std::move(x)] { cout << y << endl; }();
```

- Поэтому новые возможности языка доходят до GDB с опозданием. Скажем rvalue refs стали возможны через три года после введения в язык
- У отладчика LLDB единый парсер с Clang (прямое переиспользование кода), поэтому его возможности не запаздывают

# Отладочная информация

- DWARF -- де-факто стандарт отладочной информации в Unix-подобных системах
- PE/COFF -- де-факто стандарт отладочной информации в Windows-подобных системах
- OMF -- ранее популярный, но устаревший формат

Больше информации по DWARF  
можно найти на [dwarfstd.org](http://dwarfstd.org)

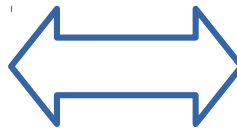


# Отладочная информация

```
$ gcc -c -save-temps -O0 -g3 -gdwarf-2 fact.c
```

fact:

```
.file 1 "fact.c"
.loc 1 3 0
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
subq $16, %rsp
movl %edi, -4(%rbp)
.loc 1 4 0
cmpl $1, -4(%rbp)
ja .L2
.loc 1 5 0
movl $1, %eax
jmp .L3
```



```
1. unsigned
2. fact (unsigned x)
3. {
4.     if (x < 2)
5.         return 1;
6.
7.     return x * fact (x-1);
8. }
```

# Отладочная информация

```
$ objdump -gdS fact.o > fact.dis
```

fact:

```
.file 1 "fact.c"
.loc 1 3 0
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
subq $16, %rsp
movl %edi, -4(%rbp)
.loc 1 4 0
cmpl $1, -4(%rbp)
ja .L2
.loc 1 5 0
movl $1, %eax
jmp .L3
```

0000000000000000 <fact>:

**unsigned fact (unsigned x) {**

```
0:      55                push    %rbp
1:      48 89 e5          mov     %rsp,%rbp
4:      48 83 ec 10       sub     $0x10,%rsp
8:      89 7d fc          mov     %edi,-0x4(%rbp)
if (x < 2)
b:      83 7d fc 01       cmpl    $0x1,-0x4(%rbp)
f:      77 07             ja      18 <fact+0x18>
return 1;
11:     b8 01 00 00 00     mov     $0x1,%eax
16:     eb 11             jmp     29 <fact+0x29>
return x * fact (x-1);
18:     8b 45 fc          mov     -0x4(%rbp),%eax
1b:     83 e8 01          sub     $0x1,%eax
1e:     89 c7             mov     %eax,%edi
20:     e8 00 00 00 00     callq   25 <fact+0x25>
25:     0f af 45 fc          imul    -0x4(%rbp),%eax
}
29:     c9             leaveq  %eax
2a:     c3             retq
```

# Соглашения о вызове

## X86 prologue

```
fact:
    .file 1 "fact.c"
    .loc 1 3 0
    .cfi_startproc
    pushq %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq %rsp, %rbp
    .cfi_def_cfa_register 6
    subq $16, %rsp
    movl %edi, -4(%rbp)
    .loc 1 4 0
```

.....

## ARMv8 prologue

```
fact:
    .file 1 "fact.c"
    .loc 1 3 0
    .cfi_startproc
    stmfd sp!, {fp, lr}
    .cfi_def_cfa_offset 8
    .cfi_offset 11, -8
    .cfi_offset 14, -4
    add fp, sp, #4
    .cfi_def_cfa 11, 4
    sub sp, sp, #8
    str r0, [fp, #-8]
    .loc 1 4 0
```

.....



# Соглашения о вызове

- **cdecl** аргументы через стек, справа налево, обратный сдвиг – caller
- **pascal** аргументы через стек, слева направо, обратный сдвиг – callee
- **fastcall** аргументы на регистрах, обратный сдвиг – callee
- **stdcall** – аргументы через стек, справа налево, обратный сдвиг – callee
- **tailcall** – вызов непосредственно перед возвратом, можно не двигать стек

# Статический анализ кода

- Массу ошибок можно поймать до того, как программа будет скомпилирована
- Большой выбор инструментов
- Есть как платные (coverity, pvs-studio) так и бесплатные (cpplint, clang-tidy) утилиты
- Проверка может работать на распространенных паттернах ошибок или делать попытки моделирования и model checking

# clang-tidy: статический анализ

```
struct B {  
    int a;  
    virtual int f();  
};  
struct D : public B {  
    int b;  
    int f() override;  
};
```

```
int use(B b) { return b.f(); }
```

Очевидна ли сейчас проблема?

# clang-tidy: статический анализ

```
struct B { int a; virtual int f(); };  
struct D : B { int b; int f() override; };
```

```
// intended: use(B &b)?  
int use(B b) { return b.f(); }
```

```
int foo () {  
    D d;  
    return use(d); // slice!  
};
```

# clang-tidy: обнаружение срезки

```
$ clang-tidy-3.8 ./slice.cpp -checks=* -- ./slice.cpp
```

```
slice.cpp:1:8: warning: Assigned value is garbage or  
undefined [clang-analyzer-core.uninitialized.Assign]
```

```
struct B { int a; virtual int f(); };  
      ^
```

```
slice.cpp:4:30: Calling implicit copy constructor for 'B'  
int foo () { D d; return use(d); };  
                ^
```

```
slice.cpp:1:8: Assigned value is garbage or undefined  
struct B { int a; virtual int f(); };
```

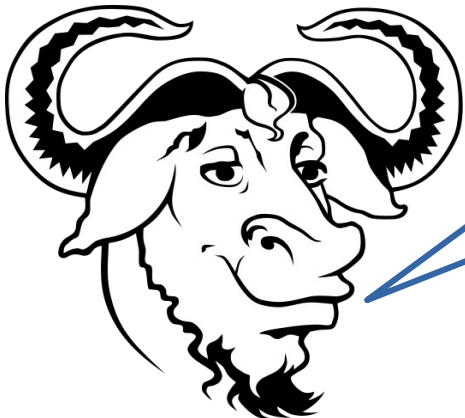


# Проверки времени исполнения

```
int foo(int* a, int len) {  
    assert ((a != NULL) && (len > 1));  
    return a[len/2];    /* ORLY? */  
}
```

- Статически такое поймать невозможно
- Выход: санитайзеры

```
$ gcc maybe-ub.c -O2 -fsanitize=address
```



Больше санитайзеров описано на  
[gcc.gnu.org](http://gcc.gnu.org)

# Valgrind: основы

- Основная проблема санитайзеров: они портят итоговый код
- Valgrind – инструмент, делающий то же самое для оригинальной программы

```
$ gcc maybe-ub.c -O2  
$ valgrind a.out  
==4468== Syscall param  
exit_group(status) contains  
uninitialised byte(s)
```



При этом valgrind может гораздо больше

# Valgrind и утечки памяти

```
int main() {  
    char *ix = malloc(10);  
    return 0;  
}
```

Эту память НИКТО  
не освободит

```
$ valgrind -tool=memcheck leak.x
```

```
==4578== HEAP SUMMARY:  
==4578==      in use at exit: 10 bytes in 1 blocks  
==4578==    total heap usage: 1 allocs, 0 frees, 10 bytes  
allocated  
==4578==  
==4578== LEAK SUMMARY:  
==4578==    definitely lost: 10 bytes in 1 blocks
```



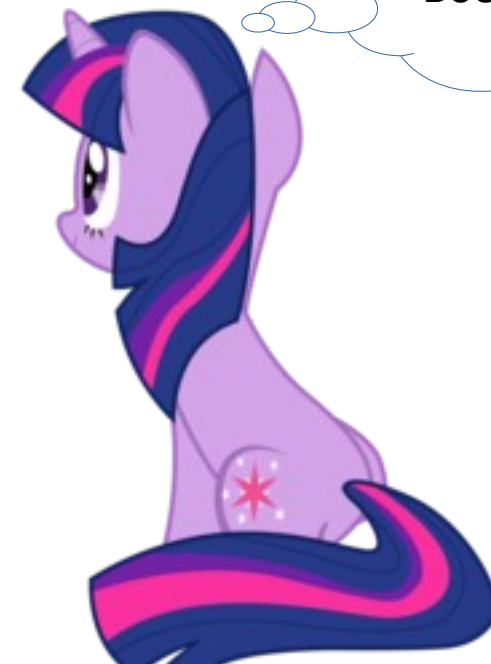
# Valgrind: ловим data races

```
#include <thread>
#include <iostream>
int x = 0;

void foo () {
    std::thread([&] { ++x; }).detach();
    ++x;
}

int main () {
    foo ();
    std::cout << x << std::endl;
    return 0;
}
```

```
$ g++ race.cpp -pthread
$ ./a.out
1
```



Вроде у меня  
всё работает

# Valgrind: ловим data races

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

int x = 0;
```

```
void foo() {
    std::cout << "x=" << x << "\n";
    ++x;
}
```

```
int main() {
    foo();
    std::cout << "x=" << x << "\n";
    return 0;
}
```

```
$ valgrind --tool=helgrind ./a.out
```

```
==4257== Possible data race during write
           of size 4 at 0x602548 by thread #1
==4257==    at 0x400DEF: foo()
==4257==    by 0x400E20: main
==4257== This conflicts with a previous write
           of size 4 by thread #2
```

# Toolchain

- Системы компиляции

- Компилятор

- Ассемблер

- Линкер

- Отладка

- **Производительность**

- Портирование



# Анализ производительности

- Сколько раз была вызвана та или иная функция?
- Сколько раз исполнялась конкретная строка кода?
- Сколько времени компилятор провёл в той или иной функции?
- Как аннотировать дугу call-graph стоимостью в процентах времени исполнения

# Анализ производительности

- Сколько раз была вызвана та или иная функция?
- Сколько раз исполнялась конкретная строка кода?
- Сколько времени компилятор провёл в той или иной функции?
- Как аннотировать дугу call-graph стоимостью в процентах времени исполнения

Code coverage



The diagram illustrates the relationship between performance analysis and code coverage. A box containing the text 'Code coverage' is positioned at the bottom. A line originates from the right side of this box, extends horizontally to the right, then turns vertically upwards, and finally turns horizontally to the left, ending with an arrowhead pointing towards the list of performance analysis questions located in the upper left area of the slide.

# Анализ покрытия: GCOV

```
$ gcc --coverage fact.c main.c -O0 -g -lgcov -o test.x  
$ ./test.x
```



```
fact.gcda (gcno)
```



```
$ gcov fact.c
```



```
fact.c.gcov
```

```
-:      1:unsigned  
5:      2:fact (unsigned x)  
-:      3:{  
5:      4:  if (x < 2)  
1:      5:      return 1;  
-:      6:  
4:      7:  return x * fact (x-1);  
-:      8:}
```

# GCOV: аннотация кода

```
static ulong counts[numbbs];  
static struct bbobj = { numbbs, &counts, "file1.c"};  
static void _GLOBAL_.I.fooBarGCOV() { __bb_init_func(&bbobj); }
```

```
void fooBar (void)  
{  
    <bb-i>  
    if (condition) {  
        <bb-j>  
    } else {  
        <bb-k>  
    }  
}
```



```
void fooBar (void)  
{  
    counts[i]++;  
    <bb-i>  
    if (condition) {  
        counts[j]++;  
        <bb-j>  
    } else {  
        <bb-k>  
    }  
}
```

# Анализ производительности

- Сколько раз была вызвана та или иная функция?
- Сколько раз исполнялась конкретная строка кода?
- Сколько времени компилятор провёл в той или иной функции?
- Как аннотировать дугу call-graph стоимостью в процентах времени исполнения



# Анализ производительности

- Сколько раз была вызвана та или иная функция?
- Сколько раз исполнялась конкретная строка кода?
- Сколько времени компилятор провёл в той или иной функции?
- Как аннотировать дугу call-graph стоимостью в процентах времени исполнения
- Какие программы достаточно репрезентативны для сравнения разных архитектур?

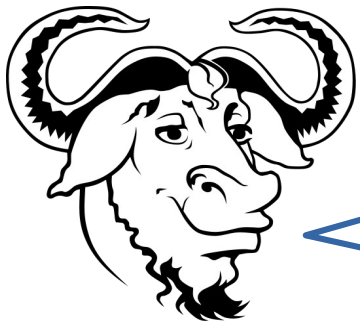
# Бенчмарки

- Dhrystone – производительность целочисленной арифметики
- Whetstone – производительность арифметики с плавающей точкой
- Coremark – пакет тестов производительности, покрывающих недостатки Dhrystone
- NAS parallel benchmarks – производительность параллельных вычислений

# Исследование dhrystone с помощью gprof

```
$ gcc -fno-inline -pg -O2 -DHZ=60 dhry_1.c dhry_2.c -o dhry
$ dhry
$ gprof dhry gmon.out --width=120
```

time	seconds	seconds	calls	ns/call	ns/call	name
24.30	0.33	0.33	10000000	32.56	32.56	Proc_8
23.18	0.64	0.31				main
13.46	0.82	0.18	30000000	6.01	6.01	Proc_7
11.22	0.97	0.15	10000000	15.03	34.07	Proc_1
9.35	1.09	0.13	30000000	4.17	4.17	Func_1
8.60	1.21	0.12	10000000	11.52	15.70	Func_2
3.74	1.26	0.05	10000000	5.01	5.01	Proc_2
3.36	1.30	0.05	10000000	4.51	7.01	Proc_6



Исходники dhrystone 2.1

<http://fossies.org/linux/privat/old/dhrystone-2.1.tar.gz>

Но это не ANSI C, их придется несколько причесать

# Профиль по call graph edges

index	% time	self	children	called	name
		0.15	0.19	10000000/10000000	main [1]
[2]	25.4	0.15	0.19	10000000	Proc_1 [2]
.....					
		0.33	0.00	10000000/10000000	main [1]
[3]	24.3	0.33	0.00	10000000	Proc_8 [3]
.....					
		0.06	0.00	10000000/30000000	main [1]
		0.06	0.00	10000000/30000000	Proc_1 [2]
		0.06	0.00	10000000/30000000	Proc_3 [8]
[4]	13.4	0.18	0.00	30000000	Proc_7 [4]

main → Proc\_1 = 25.4%

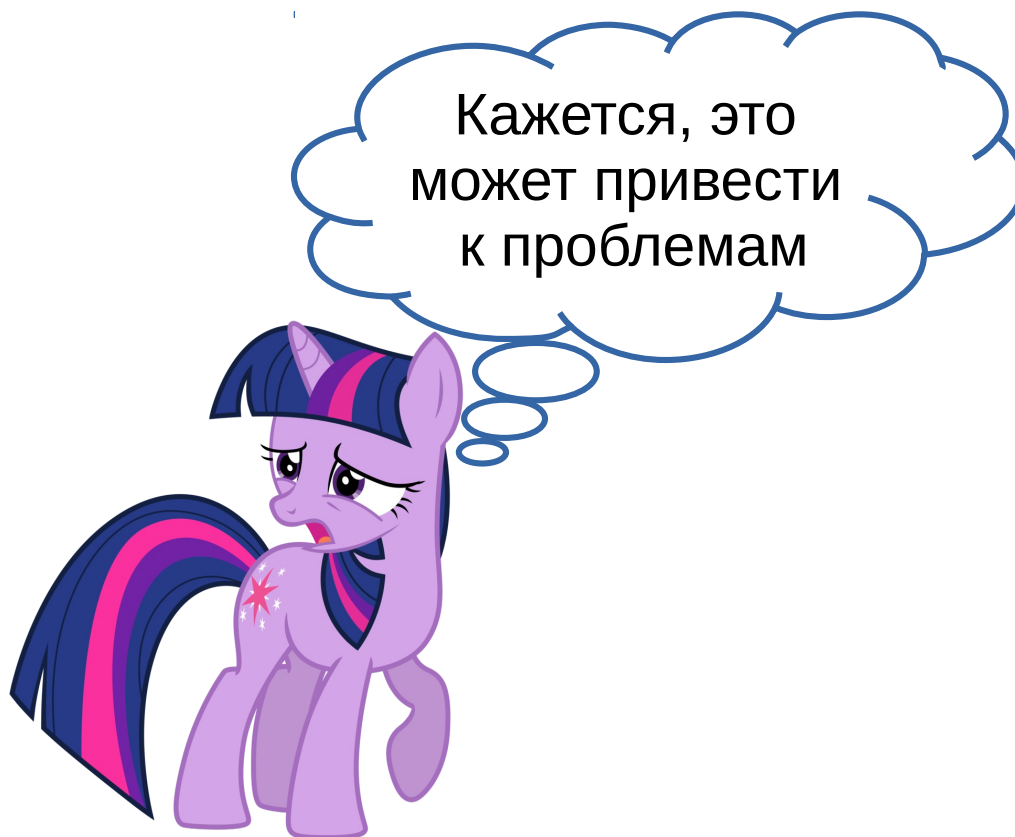
main → Proc\_8 = 24.3%

Proc\_3 → Proc\_7 = 13.4%

# Внутри gprof

- **mcount**: аннотация кода количеством ВЫЗОВОВ
- **profil**: сто раз в секунду информация о текущей функции

```
fact:  
pushq    %rbp  
movq     %rsp, %rbp  
call     mcount  
movl     $1, %eax  
cmpl     $1, %edi  
ja       .L3  
.....
```



# Проблемы gprof

```
void work(int n) {  
    volatile int i=0; //don't optimize away  
    while(i++ < n);  
}  
  
void easy() { work(1000); }  
void hard() { work(1000*1000); }  
int main() { easy(); hard(); }
```

Какой аннотации дуг мы ожидаем?

# Проблемы gprof

```
void work(int n) {  
    volatile int i=0; //don't optimize away  
    while(i++ < n);  
}  
  
void easy() { work(1000); }  
void hard() { work(1000*1000); }  
int main() { easy(); hard(); }
```

Примерно такой:

main → hard = 99.9%

main → easy = 0.01%

Но на самом деле...

# Проблемы gprof

```
void work(int n) {  
    volatile int i=0; //don't optimize away  
    while(i++ < n);  
}  
  
void easy() { work(1000); }  
void hard() { work(1000*1000); }
```

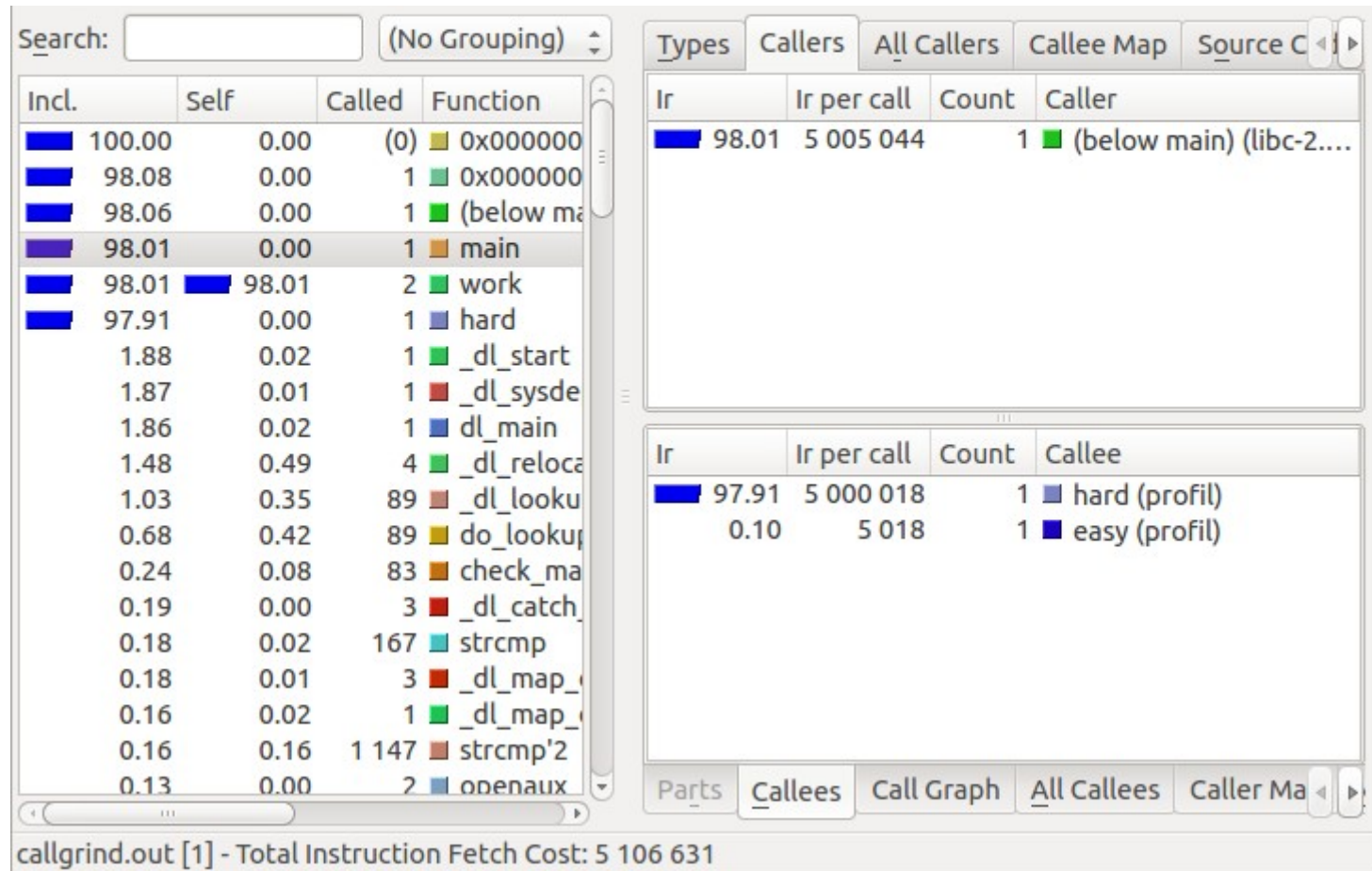
index	% time	self	children	called	name
		0.00	1.09	1/1	main [2]
[3]	50.0	0.00	1.09	1	easy [3]
-----					
		0.00	1.09	1/1	main [2]
[4]	50.0	0.00	1.09	1	hard [4]



# Альтернатива: callgrind

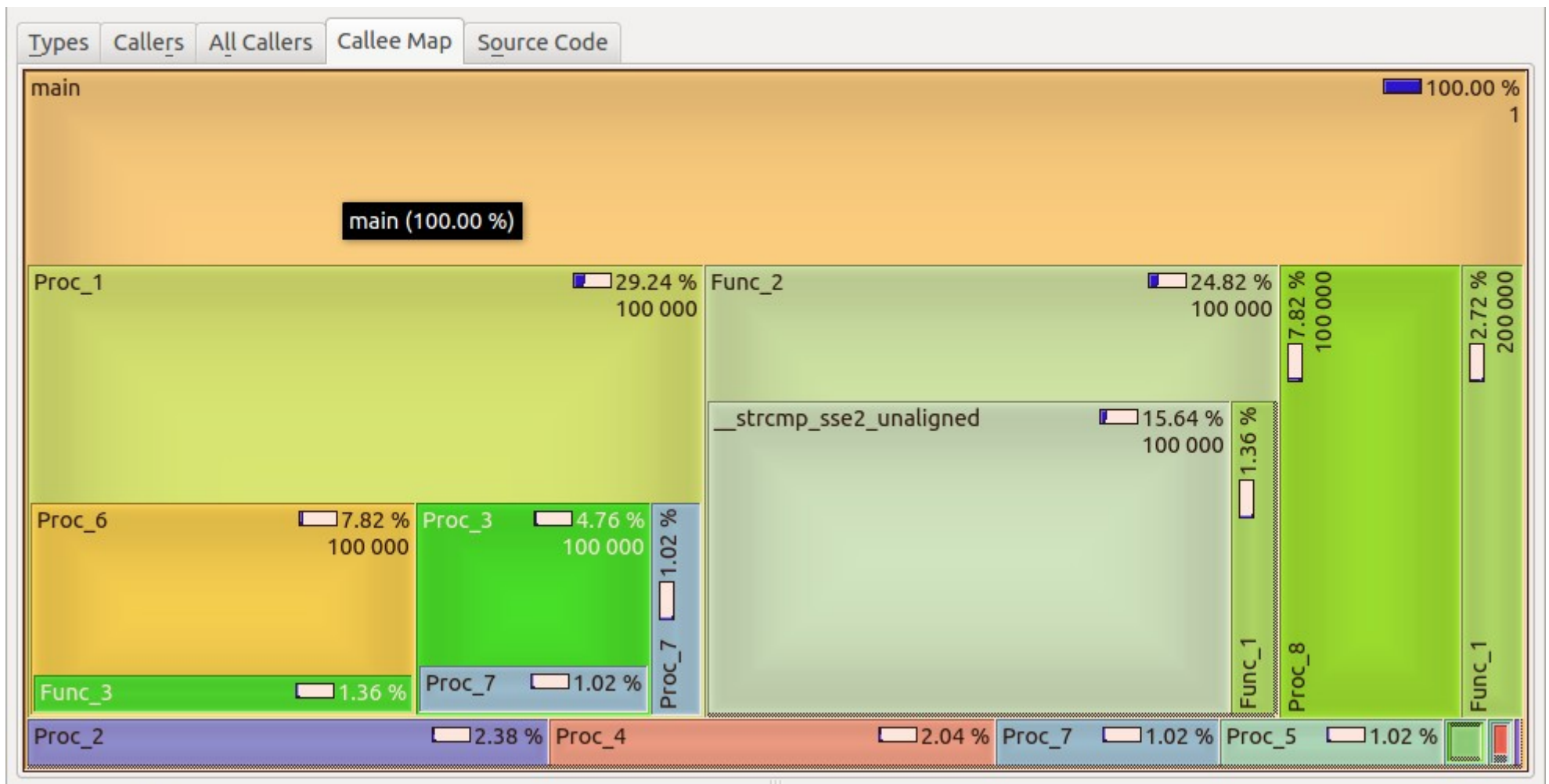
```
$ gcc -O2 profil.c -o profil
$ valgrind --tool=callgrind ./profil
$ kcachegrind
```

Гораздо  
лучше



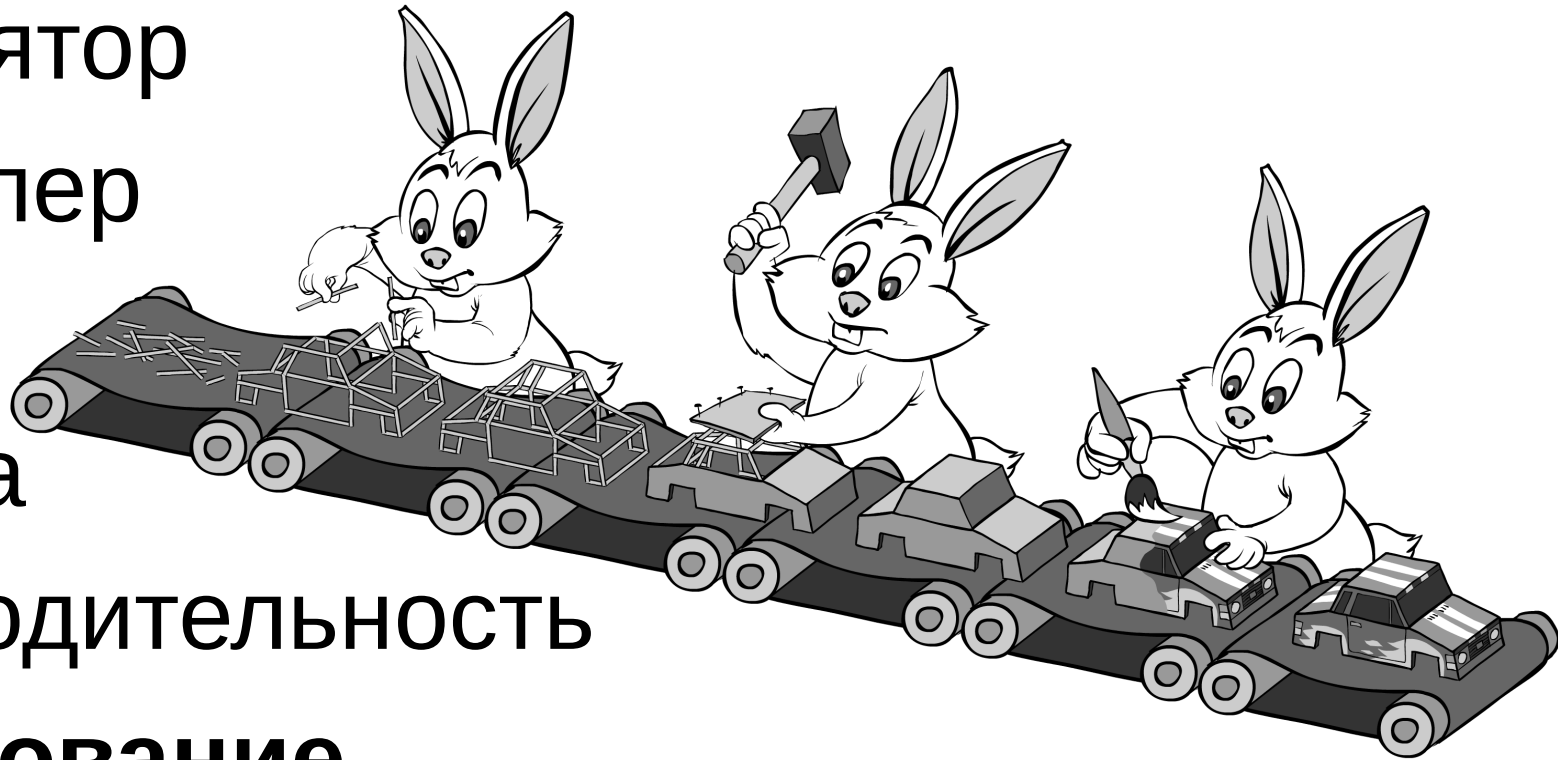
# Исследование dhrystone с помощью callgrind

```
$ gcc -fno-inline -O2 -DHZ=60 dhry_1.c dhry_2.c -o dhry
$ valgrind --tool=callgrind ./dhry
$ kcachegrind
```



# Toolchain

- Системы компиляции
- Компилятор
- Ассемблер
- Линкер
- Отладка
- Производительность
- **Портирование**



# Портирование toolchain

- Бэкенд компилятора (GCC, LLVM)
- Бинарные утилиты (as, ld, ...)
- Стандартные библиотеки
- Отладчики (GDB, ...)

Новым архитектурам  
требуется средства  
разработки



# Перед портированием

Определить набор инструкций (ISA)

Выбрать кодировку для инструкций

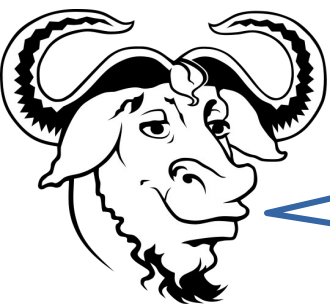
Договориться о синтаксисе ассемблера

Документировать вызов функций (ABI)

Выбрать формат выходных файлов

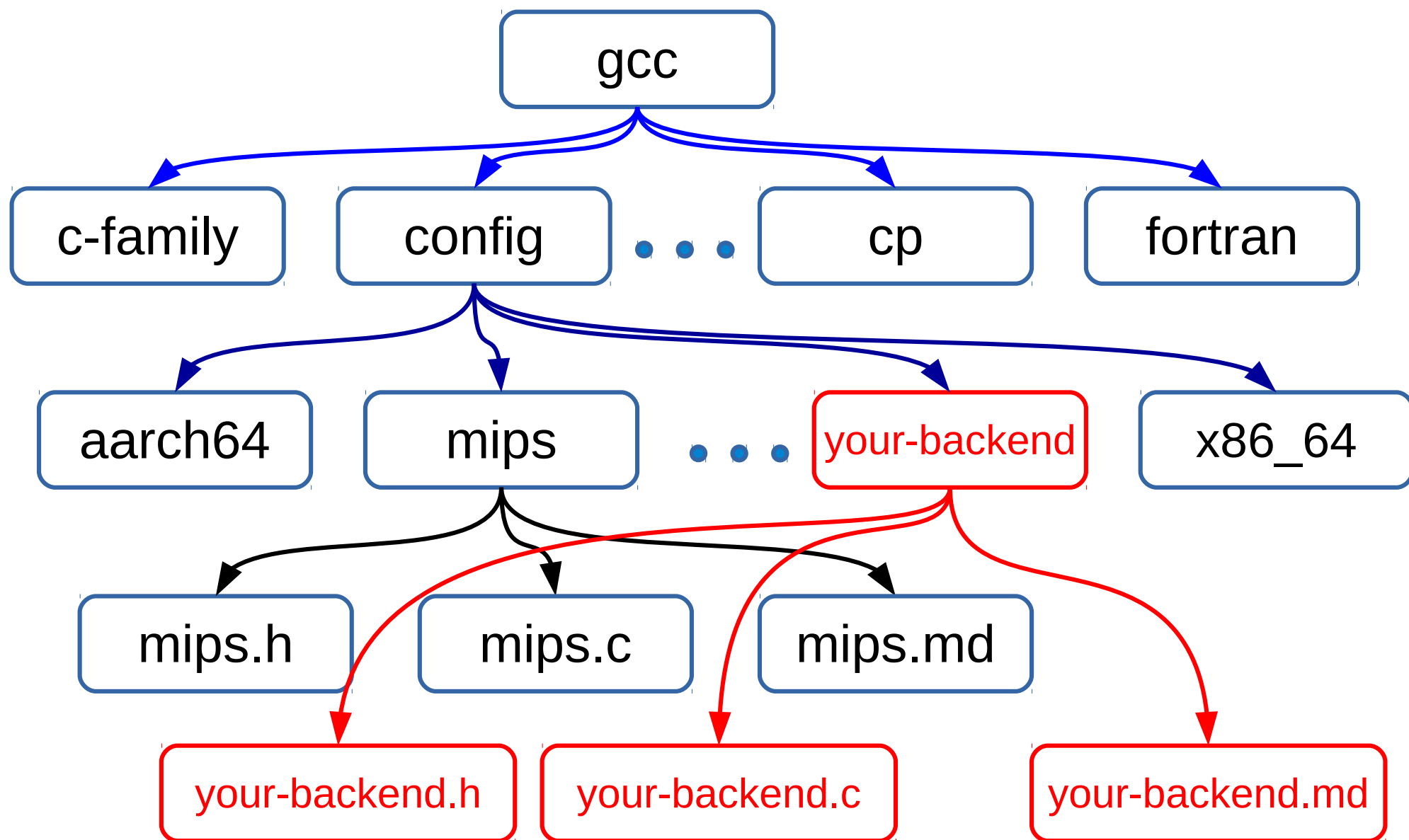
# Application binary interface

- Как организована память при исполнении программы
- Как происходит вызов функций и передача параметров
- На какие системные интерфейсы могут полагаться программы



ABI для архитектуры i386:  
[www.sco.com/developers/devspecs/abi386-4.pdf](http://www.sco.com/developers/devspecs/abi386-4.pdf)

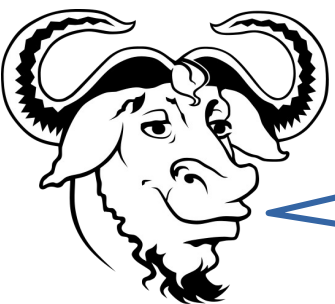
# Внутри GCC: структура файлов



# Внутри GCC: паттерны md

```
(define_insn "*addsi3_extended"
```

```
[(set_attr "alu_type" "add")  
 (set_attr "mode" "SI")])
```



Больше информации про GCC internals  
[gcc.gnu.org/onlinedocs/gccint/](http://gcc.gnu.org/onlinedocs/gccint/)



# Внутри GCC: паттерны md

```
(define_insn "*addsi3_extended"  
  [(set (match_operand:DI 0 "register_operand" "=d,d")  
    (sign_extend:DI  
      (plus:SI (match_operand:SI 1 "register_operand" "d,d")  
        (match_operand:SI 2 "arith_operand" "d,Q")))))]
```

```
[(set_attr "alu_type" "add")  
 (set_attr "mode" "SI")]
```

```
r0:DI = sext:DI (r1:SI + a2:SI)
```

# Внутри GCC: паттерны md

```
(define_insn "*addsi3_extended"  
  [(set (match_operand:DI 0 "register_operand" "=d,d")  
        (sign_extend:DI  
          (plus:SI (match_operand:SI 1 "register_operand" "d,d")  
                   (match_operand:SI 2 "arith_operand" "d,Q"))))]  
  "TARGET_64BIT && !TARGET_MIPS16"
```

```
[(set_attr "alu_type" "add")  
 (set_attr "mode" "SI")]
```

```
r0:DI = sext:DI (r1:SI + a2:SI) iff condition
```

# Внутри GCC: паттерны md

```
(define_insn "*addsi3_extended"  
  [(set (match_operand:DI 0 "register_operand" "=d,d")  
        (sign_extend:DI  
          (plus:SI (match_operand:SI 1 "register_operand" "d,d")  
                   (match_operand:SI 2 "arith_operand" "d,Q"))))]  
  "TARGET_64BIT && !TARGET_MIPS16"  
  "@  
  addu\t%0,%1,%2  
  addiu\t%0,%1,%2"  
  [(set_attr "alu_type" "add")  
   (set_attr "mode" "SI")])
```

\*addsi3\_extended (r0, r1, a2)



"addu %0,%1,%2"

# Внутри GCC: преобразования инструкций

```
(define_split
  [(set (match_operand:DI 0 "register_operand")
        (and:DI (match_operand:DI 1 "register_operand")
                  (const_int 4294967295)))]
  "TARGET_64BIT && reload_completed"
  [(set (match_dup 0)
        (ashift:DI (match_dup 1) (const_int 32)))
   (set (match_dup 0)
        (lshiftrt:DI (match_dup 0) (const_int 32)))])
```

`r0:DI = r1:DI and (-1)`



`r0:DI = r1:DI << 32`  
`r0:DI = r0:DI >> 32`

# Список литературы

Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman  
"Compilers: Principles, Techniques, and Tools, 2nd Edition"

Steven Muchnick  
"Advanced Compiler Design and Implementation"

John R. Levine  
"Linkers and Loaders"

GCC Internals

Binutils Internals



# Экзаменационные вопросы

- Препроцессор и фронтенд компилятора
- Бэкенд компилятора
- Ассемблер
- Линкер: статические библиотеки
- Линкер: динамические библиотеки
- Отладка и статический анализ кода
- Анализ производительности
- Портирование компилятора