Лекция 0x12

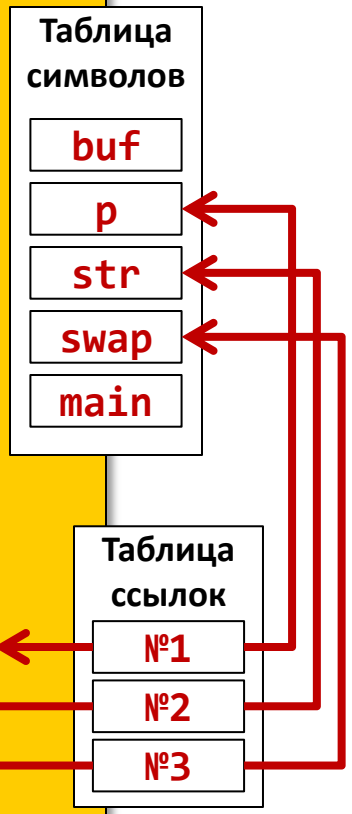
17 апреля

Построение объектного кода

```
int buf[2] = {1, 2};
char str[] = "Hello";
char *p    = "world";

int swap(char*, char*);

int main(int argc, char* argv[]) {
   if (argc > 1) {
      swap(str, p);
   }
   return 0;
}
```

```
   push   eax
   push   dword [p]
   push   str
   call   swap
   add    esp,0x10
.L2:
   mov    ecx,dword [ebp-0x4]
   xor    eax,eax
   leave
   lea    esp,[ecx-0x4]
   ret
```

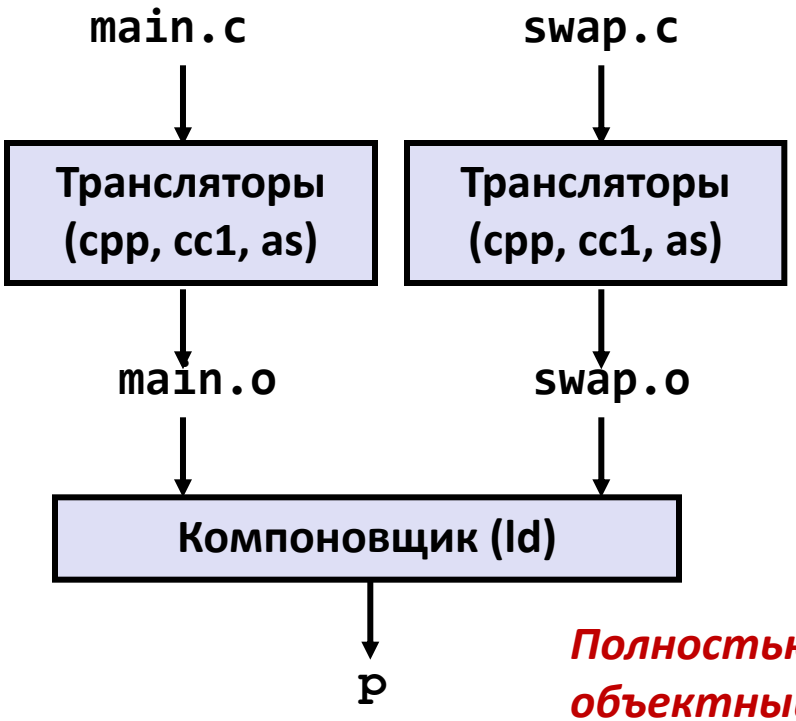
```
00000000 <main>:
   0:   8d 4c 24 04
   4:   83 e4 f0
   7:   ff 71 fc
   a:   55
   b:   89 e5
   d:   51
   e:   83 ec 04
  11:   83 39 01
  14:   7e 15
  16:   50
  17:   50
  18:   ff 35 ** ** ** **
  1e:   68 ** ** ** **
  23:   e8 ** ** ** **
  28:   83 c4 10

  2b:   8b 4d fc
  2e:   31 c0
  30:   c9
  31:   8d 61 fc
  34:   c3
```



Работоспособный код указанных команд не может быть построен ассемблером . Компоновщик окончательно заполнит в командах байты, относящееся к ссылкам.

Статическая компоновка

- Программа транслируется и компонуется драйвером компилятора:
  - `unix> gcc –O3 -g -o p main.c swap.c`
  - `unix> ./p`



Файлы с исходным кодом

Независимо друг от друга откомпилированные файлы с перемещаемым объектным кодом

Полностью скомпонованный исполняемый объектный файл (содержит код и данные всех функций определенные в `main.c` и `swap.c`)

Почему нужен компоновщик?

- Причина 1: Модульность программы
  - Программа может быть организована как набор небольших файлов с исходным кодом, а не один монолитный файл.
  - Есть возможность организовывать библиотеки функций, являющихся общими для разных программ
    - например, библиотека математических функций, стандартная библиотека языка Си

Почему нужен компоновщик?

- Причина 2: Эффективность
  - Время: Раздельная компиляция
    - Меняем код в одном файле, компилируем только его, повторяем компоновку
    - Нет необходимости повторять компиляцию остальных файлов с исходным кодом.
  - Место на диске: Библиотеки
    - Общие функции можно объединить в одном файле...
    - Исполняемые файлы и образ программы в памяти содержит только те функции, которые действительно используются.

Что делает компоновщик?

- Шаг 1. Разрешение символов
  - В ассемблерной программе определяют и используют *символы* (переменные и функции):
    - `void swap() {…}  /* определение символа swap */`
    - `swap(str, p);  /* ссылки на символы */`
    - `int *xp = &x;  /* определение символа xp, ссылка на x */`
  - Определения символов сохраняются в таблице символов.
    - Таблица символов – массив структур
    - Каждая запись содержит имя, размер, позицию символа.
  - Компоновщик устанавливает связь каждой ссылки на символ с единственным определением символа.

```
typedef struct {
   int name;       /* Смещение в таблице строк */
   int value;      /* Смещение в секции или виртуальный адрес */
   int size;       /* Размер в байтах */
   char type:4,    /* Данные, код, секция, имя файла */
   binding:4;      /* Локальный или глобальный символ */
   char reserved; /* Не используется */
   char section;  /* Номер заголовка секции, ABS, UNDEF, COMMON */
} Elf_Symbol;
```

Что делает компоновщик?

- Шаг 2. Перемещение
  - Несколько объявлений секций кода и данных объединяются в единые секции
  - Символы перемещаются с их относительных позиций в `.o` файлах на абсолютные адреса в исполняемом файле.
  - Обновляются (перебазируются) все ссылки на символы, согласно их новым позициям.

```
typedef struct {
   int offset;    /* Смещение перебазируемой ссылки в секции */
   int symbol:24, /* Номер символа в таблице */
   type:8;        /* Тип перебазирования */
} Elf32_Rel;
```

Три типа объектных файлов (модулей)

- Перемещаемые объектные файлы (`.o`-файлы)
  - Содержит код и данные в форме, позволяющей проводить компоновку с другими перемещаемым объектными файлами.
    - Каждый `.o`-файл производится из **одного** файла с исходным кодом (`.c`-файла)
- Исполняемые объектные файлы
  - Содержит код и данные в такой форме, что их можно напрямую копировать в память и запускать выполнение программы.

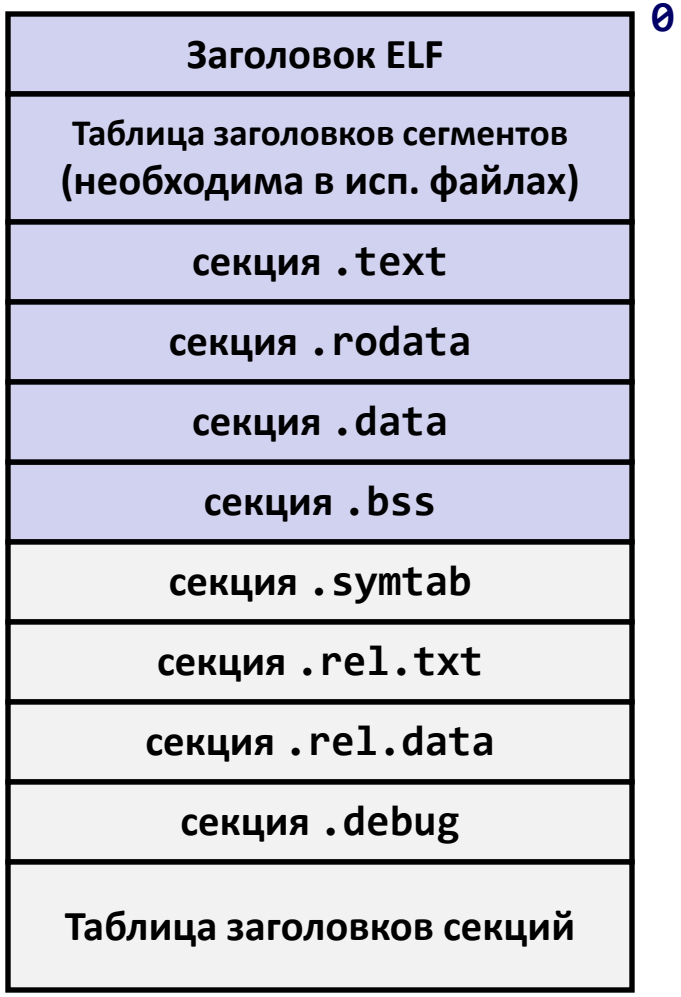
- Разделяемые объектные файлы (`.so`-файлы)
  - Особый вид перемещаемого объектного файла, который может быть загружен в память и скомпонован с программой динамически, во время ее загрузки и во время работы.
  - Windows - Dynamic Link Libraries (DLL)

Executable and Linkable Format (ELF)

- Стандартный бинарный формат объектных файлов
- Был предложен в AT&T System V Unix
  - Позже был поддержан в BSD и Linux
- Единый формат для
  - Перемещаемых объектных файлов (`.o`),
  - Исполняемых объектных файлов
  - Разделяемых объектных файлов (`.so`)

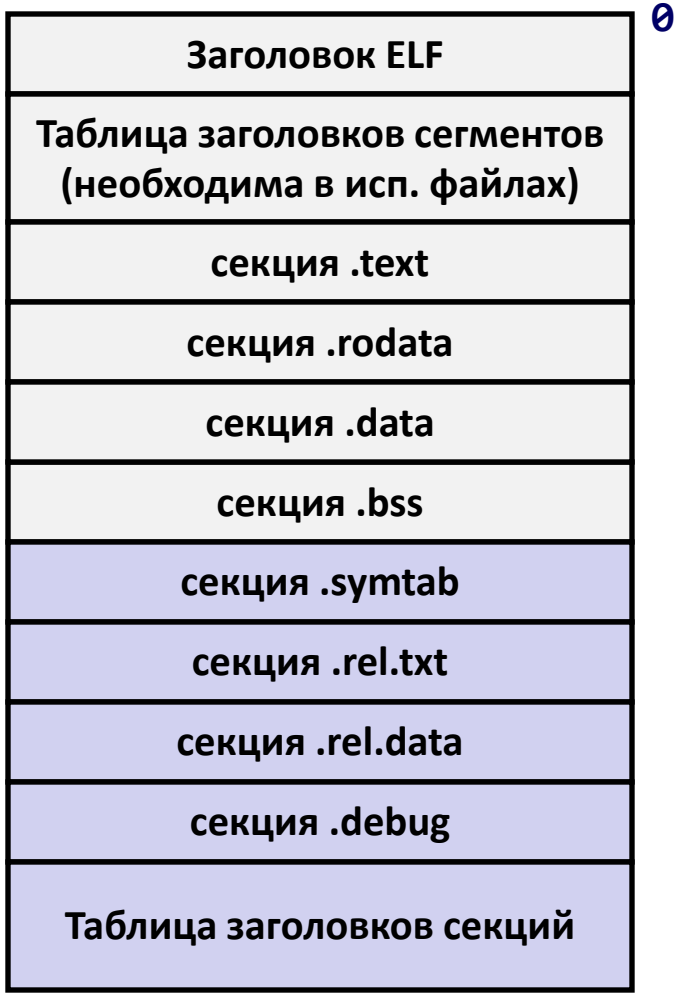
Формат ELF файла

- Заголовок Elf
  - Размер машинного слова, порядок байт, тип файла (`.o`, исп., `.so`), и др.
- Таблица заголовков сегментов
  - Размер страницы, сегменты виртуальной памяти, размеры сегментов.
- Секция `.text`
  - код
- Секция `.rodata`
  - Данные, доступные только на чтение: таблицы переходов, константы
- Секция `.data`
  - Инициализированные глобальные переменные
- Секция `.bss`
  - Неинициализированные глобальные переменные
  - У секции есть заголовок, на сама секция не занимает места



Формат ELF файла (продолжение)

- Секция `.symtab`
  - Таблица символов
  - Имена функций и статических переменных
  - Имена секций
- Секция `.rel.text`
  - Данные для перемещения ссылок в секции `.text`
  - Адреса обновляемых операндов в двоичном коде инструкций
- Секция `.rel.data`
  - Данные для перемещения ссылок в секции `.data`
  - Адреса глобальных переменных, инициализированных ссылками на внешние функции или глобальные переменные
- Секция `.debug`
  - Данные для символьного отладчика (`gcc -g`)
- Таблица заголовков секций
  - Смещения и размеры каждой секции



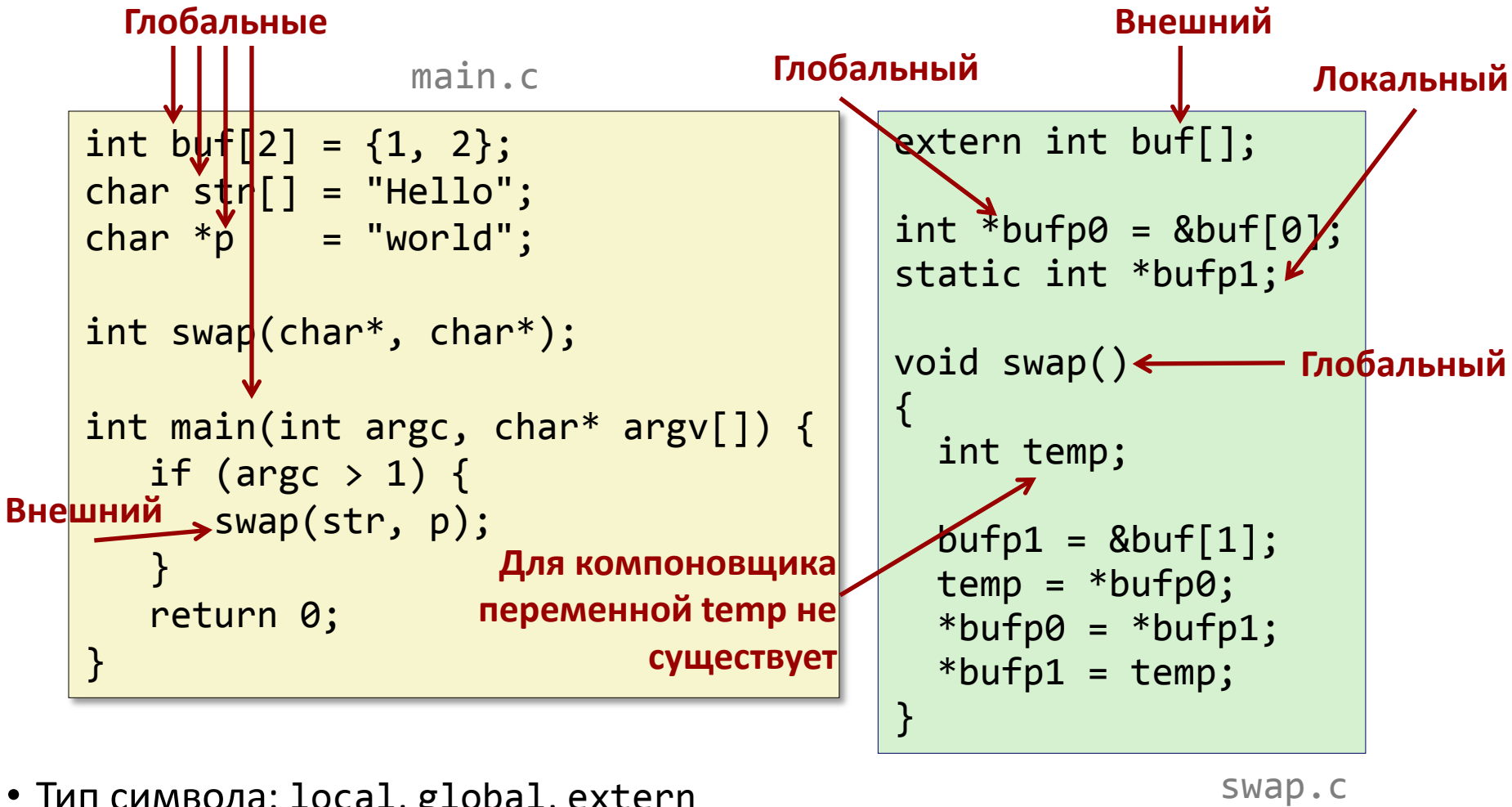
Символы в процессе компоновки

- Глобальные символы
  - Символы определенные в одном модуле таким образом, что их можно использовать в других модулях.
  - Например: не-`static` Си-функции и не-`static` глобальные переменные.

- Внешние (неопределенные) символы
  - Глобальные символы, которые используются в модуле, но определены в каком-то другом модуле.

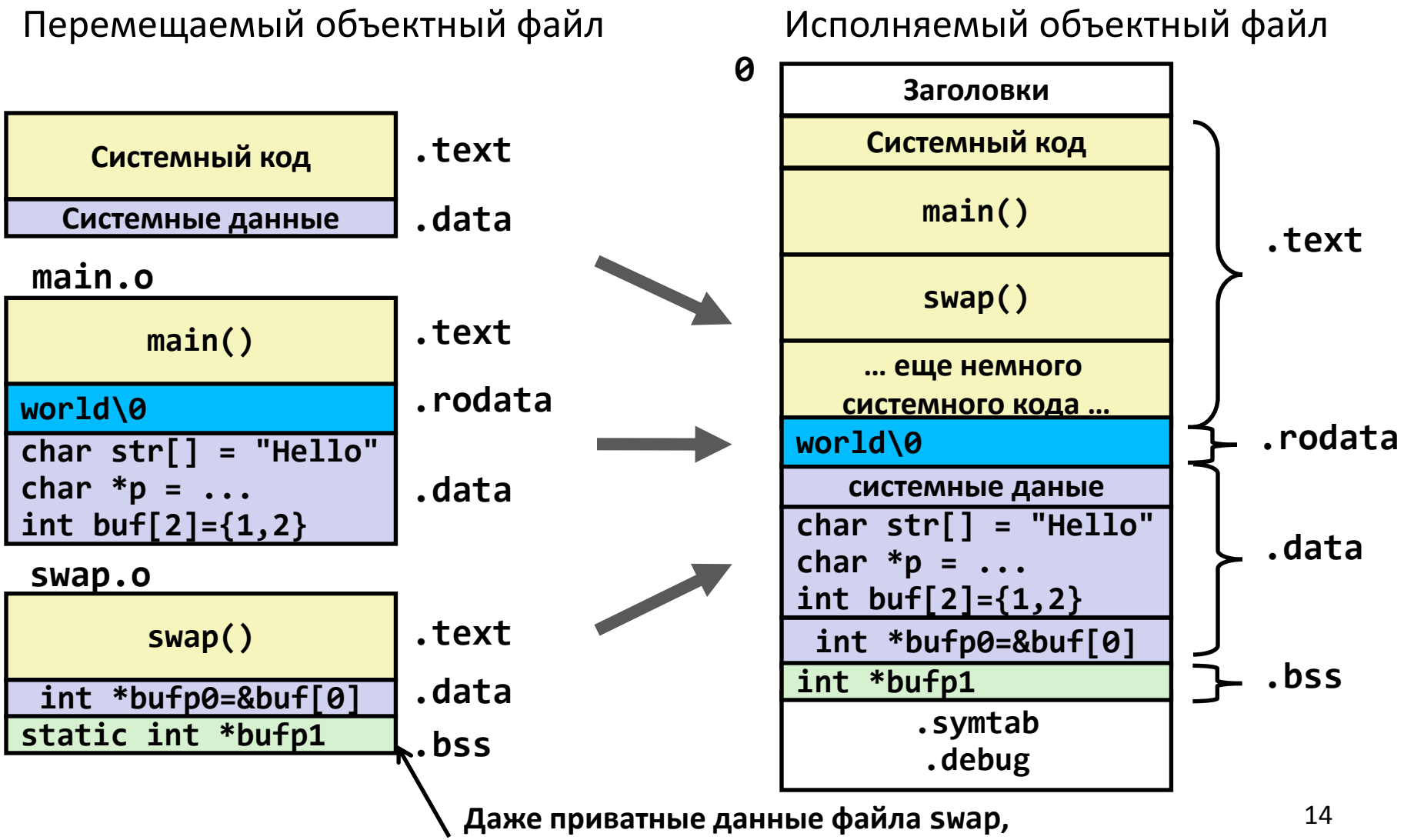
- Локальные символы
  - Символы определены и используются исключительно в одном модуле.
  - Например: Си-функции и переменные, определенные с модификатором `static`.
  - **Локальные символы *не являются* локальными переменными Си-программы**

Задача: разрешение символов



- Тип символа: `local`, `global`, `extern`
- Присутствует ли символ в `.symtab`?
- Модуль, в котором символ определен?
- Размер?
- Смещение внутри секции?

Перемещение кода и данных



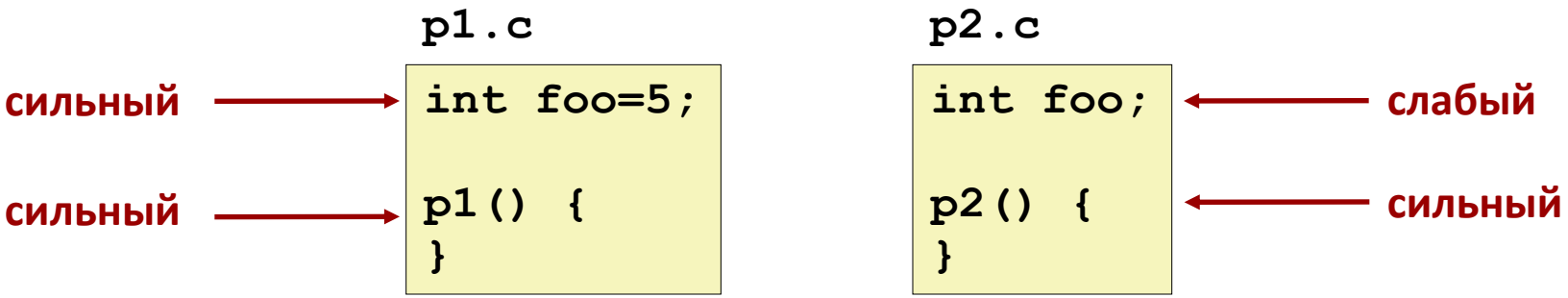
Предварительное определение переменных

```
int i1 = 10;          /* определение, внешнее связывание */
static int i2 = 20;   /* определение, внутреннее связывание */
extern int i3 = 30;   /* определение, внешнее связывание */
int i4;               /* предварительное определение, внешнее связывание */
static int i5;        /* предварительное определение, внутреннее связывание */

int i1;               /* корректное предварительное определение */
int i2;               /* ошибка, неувязка с уже заданным связыванием */
int i3;               /* корректное предварительное определение */
int i4;               /* корректное предварительное определение */
int i5;               /* ошибка, неувязка с уже заданным связыванием */
```

- Связывание имен
  - Внешнее: переменная доступна во всех единицах трансляции
  - Внутреннее: переменная доступна только в текущей единице трансляции
- Определение
  - В отсутствии инициализатора, определение трактуется как «предварительное» (tentative definition). В отсутствии других определений в данной единице трансляции переменной присваивается нулевое значение.

Сильные и слабые символы

- Каждый символ в программе либо «сильный», либо «слабый»
  - ***Сильные***: функции и инициализированные глобальные переменные
  - ***Слабые***: неинициализированные глобальные переменные



Правила работы с символами

- Правило 1: Одинаковые сильные символы запрещены
  - Каждый элемент может быть определен только один раз
  - В противном случае ошибка компоновки
- Правило 2: Один сильный символ и несколько слабых – выбираем сильный символ
  - Ссылки на слабые символы заменяются ссылками на сильный символ
- Правило 3: Если несколько слабых символов, выбираем произвольный
  - Поведение можно поменять: gcc –fno-common

```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

```
snoop@jezek:~/samples/2017/linking$ gcc -c swap.c
snoop@jezek:~/samples/2017/linking$ readelf -s swap.o

Symbol table '.symtab' contains 12 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
...
    10: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND buf
...
```

```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;
int buf[];
void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

- COMMON-символы в перемещаемом коде не относят к какой-либо реально существующей секции
- Место в памяти для таких символов выделяется при построении исполняемого кода в общей .bss секции
- При «слиянии» COMMON-символов есть возможность проверить их размер

```
snoop@jezek:~/samples/2017/linking$ gcc -c swap.c
snoop@jezek:~/samples/2017/linking$ readelf -s swap.o

Symbol table '.symtab' contains 12 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
...
    10: 00000004     4 OBJECT  GLOBAL DEFAULT  COM buf
...
```

- Отказ от механизма COMMON-символов приводит к «принудительному» выделению памяти в для них в секции .bss
- В силу особенностей языка Си и правил построения исполняемого кода, программа выделит в памяти место для каждого предварительно определенного «экземпляра» переменной
- Последствия?

```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;
int buf[];
void swap()
{
```

```
snoop@jezek:~/samples/2017/linking$ gcc -fno-common -c swap.c
snoop@jezek:~/samples/2017/linking$ readelf -s swap.o

Symbol table '.symtab' contains 12 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
...
    10: 00000000     4 OBJECT  GLOBAL DEFAULT    5 buf
...
snoop@jezek:~/samples/2017/linking$ readelf -S swap.o
There are 13 section headers, starting at offset 0x24c:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
...
  [ 5] .bss              NOBITS          00000000 000070 000008 00  WA  0   0  4
...
```

Задача

`int x;` `p1() {}`	`p1() {}`	Ошибка компоновки: два сильных символа (**p1**)
`int x;` `p1() {}`	`int x;` `p2() {}`	Ссылки на **x** будут ссылаться на один и тот же неинициализированный **int**. Но какой?

`int x;`
`int y;`
`p1() {}`
	`double x;`
	`p2() {}`
		Запись в **x** (**p2**) может поменять **y**!

`int x=7;`
`int y=5;`
`p1() {}`
	`double x;`
	`p2() {}`
		Запись в **x** (**p2**) ***обязательно поменяет*** **y**!

`int x=7;` `p1() {}`	`int x;` `p2() {}`	Ссылка на **x** будет ссылаться на один и тот же инициализированный **int**.

Наихудший сценарий: два одинаковые «слабые» структуры,
Откомпилированные разными компиляторами с разными правилами выравнивания.

Глобальные переменные

- Следует избегать, если только есть такая возможность

- В противном случае
  - Используйте **`static`** если это возможно
  - Если определяете глобальную переменную, инициализируйте ее
  - Используйте **`extern`** если ссылаетесь на внешнюю глобальную переменную

hello1.c

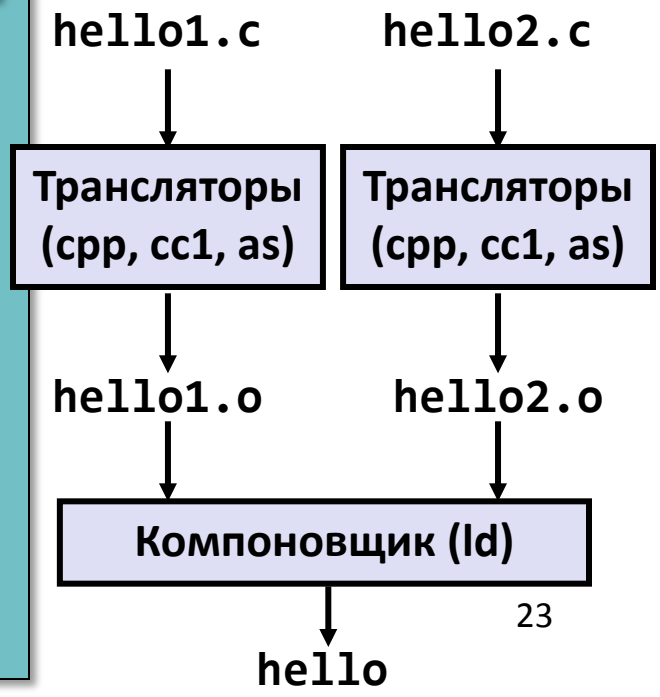
```
extern void func();

char *buf = "Hello, world!\n";

int main() {
  int ret_code = 0;
  func();
  return ret_code;
}
```

hello2.c

```
#include <stdio.h>

extern char* buf;

void func() {
   printf("%s", buf);
}
```

Makefile

```
all: hello

hello: hello1.o hello2.o
   gcc -Xlinker -M -o hello hello1.o hello2.o

hello1.o: hello1.c
   gcc -c -o hello1.o hello1.c

hello2.o: hello2.c
   gcc -c -o hello2.o hello2.c

clean:
   rm -f hello hello1.o hello2.o
```



hello1.c

```
extern void func();

char *buf = "Hello, world!\n";

int main() {
  int ret_code = 0;
  func();
  return ret_code;
}
```

```
snoop@earth:~/samples/2014$ nm hello1.o
00000000 D buf
         U func
00000000 T main
```

hello1.c

```
extern void func();

char *buf = "Hello, world!\n";

int main() {
  int ret_code = 0;
  func();
  return ret_code;
}
```

```
snoop@earth:~/samples/2014$ readelf -s hello1.o

Symbol table '.symtab' contains 11 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS hello1.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    5
     5: 00000000     0 SECTION LOCAL  DEFAULT    6
     6: 00000000     0 SECTION LOCAL  DEFAULT    8
     7: 00000000     0 SECTION LOCAL  DEFAULT    7
     8: 00000000     4 OBJECT  GLOBAL DEFAULT    3 buf
     9: 00000000    28 FUNC    GLOBAL DEFAULT    1 main
    10: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND func
```

hello1.c

```
extern void func();

char *buf = "Hello, world!\n";

int main() {
  int ret_code = 0;
  func();
  return ret_code;
}
```

```
snoop@earth:~/samples/2014$ readelf -r hello1.o

Relocation section '.rel.text' at offset 0x388 contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000012  00000a02 R_386_PC32        00000000   func

Relocation section '.rel.data' at offset 0x390 contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000000  00000501 R_386_32          00000000   .rodata
```

Что будет, если поменять код?

```
...
char buf[] = "Hello, world!\n";
...
```

Напоминание
Диалект Intel-синтаксиса, который используют программы binutils, отличается от диалекта, поддерживаемого nasm.

```
snoop@earth:~/samples/2014$ objdump -M intel -dr hello1.o

hello1.o:     file format elf32-i386


Disassembly of section .text:

00000000 <main>:
   0:   55                      push   ebp
   1:   89 e5                   mov    ebp,esp
   3:   83 e4 f0                and    esp,0xfffffff0
   6:   83 ec 10                sub    esp,0x10
   9:   c7 44 24 0c 00 00 00    mov    DWORD PTR [esp+0xc],0x0
  10:   00
  11:   e8 fc ff ff ff          call   12 <main+0x12>
                        12: R_386_PC32  func
  16:   8b 44 24 0c             mov    eax,DWORD PTR [esp+0xc]
  1a:   c9                      leave
  1b:   c3                      ret
```

```
snoop@earth:~/samples/2014$ readel

Relocation section '.rel.text' at
 Offset     Info    Type
00000012  00000a02 R_386_PC32

Relocation section '.rel.data' at
 Offset     Info    Type
00000000  00000501 R_386_32
```

```
snoop@earth:~/samples/2014$ objdump -s -j .data hello1.o

hello1.o:     file format elf32-i386

Contents of section .data:
 0000 00000000                             ....


snoop@earth:~/samples/2014$ objdump -s -j .rodata hello1.o

hello1.o:     file format elf32-i386

Contents of section .rodata:
 0000 48656c6c 6f2c2077 6f726c64 210a00    Hello, world!..
```

```
snoop@earth:~/samples/2014$ readelf -r hello1.o

Relocation section '.rel.text' at offset 0x388 contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000012  00000a02 R_386_PC32        00000000   func

Relocation section '.rel.data' at offset 0x390 contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000000  00000501 R_386_32          00000000   .rodata
```

hello2.c

```
#include <stdio.h>

extern char* buf;

void func() {
   printf("%s", buf);
}
```

```
snoop@earth:~/samples/2014$ nm hello2.o
         U buf
00000000 T func
         U printf
```

hello2.c

```
#include <stdio.h>

extern char* buf;

void func() {
   printf("%s", buf);
}
```

```
snoop@earth:~/samples/2014$ readelf -s hello2.o

Symbol table '.symtab' contains 11 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS hello2.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    5
     6: 00000000     0 SECTION LOCAL  DEFAULT    7
     7: 00000000     0 SECTION LOCAL  DEFAULT    6
     8: 00000000    31 FUNC    GLOBAL DEFAULT    1 func
     9: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND buf
    10: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
```

hello2.c

```
#include <stdio.h>

extern char* buf;

void func() {
   printf("%s", buf);
}
```

```
snoop@earth:~/samples/2014$ readelf -r hello2.o

Relocation section '.rel.text' at offset 0x354 contains 3 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000008  00000901 R_386_32          00000000   buf
0000000d  00000501 R_386_32          00000000   .rodata
00000019  00000a02 R_386_PC32        00000000   printf
```

```
snoop@earth:~/samples/2014$ objdump -M intel -dr hello2.o

hello2.o:     file format elf32-i386


Disassembly of section .text:

00000000 <func>:
   0:   55                      push   ebp
   1:   89 e5                   mov    ebp,esp
   3:   83 ec 18                sub    esp,0x18
   6:   8b 15 00 00 00 00       mov    edx,DWORD PTR ds:0x0
                        8: R_386_32     buf
   c:   b8 00 00 00 00          mov    eax,0x0
                        d: R_386_32     .rodata
  11:   89 54 24 04             mov    DWORD PTR [esp+0x4],edx
  15:   89 04 24                mov    DWORD PTR [esp],eax
  18:   e8 fc ff ff ff          call   19 <func+0x19>
                        19: R_386_PC32  printf
  1d:   c9                      leave
  1e:   c3                      ret
```

```
snoop@earth:~/samples/2014$ readelf -r hell

Relocation section '.rel.text' at offset 0x
 Offset     Info    Type            Sym.Val
00000008  00000901 R_386_32          000000
0000000d  00000501 R_386_32          000000
00000019  00000a02 R_386_PC32        000000
```

```
snoop@earth:~/samples/2014$ gcc -Xlinker -M -o hello hello1.o hello2.o
...
 .text          0x0000000080483e4       0x1c hello1.o
                0x0000000080483e4              main
 .text          0x0000000008048400      0x1f hello2.o
                0x0000000008048400             func
...
 .rodata        0x0000000080484e0        0xf hello1.o
 .rodata        0x0000000080484ef        0x3 hello2.o
...
 .data          0x000000000804a014       0x4 hello1.o
                0x000000000804a014             buf
 .data          0x000000000804a018       0x0 hello2.o
```

```
snoop@earth:~/samples/2014$ nm hello1.o
00000000 D buf
         U func
00000000 T main
```

```
snoop@earth:~/samples/2014$ nm hello2.o
         U buf
00000000 T func
         U printf
```

Стандартная библиотека языка Си *printf.o*

```
snoop@earth:~/samples/2014$ gcc -Xlinker -M -o hello hello1.o hello2.o
...
 .text          0x00000000080483e4       0x1c hello1.o
                0x00000000080483e4                main
 .text          0x0000000008048400       0x1f hello2.o
                0x0000000008048400                func
...
 .rodata        0x00000000080484e0        0xf hello1.o
 .rodata        0x00000000080484ef        0x3 hello2.o
...
 .data          0x000000000804a014        0x4 hello1.o
                0x000000000804a014                buf
 .data          0x000000000804a018        0x0 hello2.o
```

- Правила пересчета для значений ссылок
  - Тип перебазирования R_386_32
    новое значение ссылки = S + A
  - Тип перебазирования R_386_PC32
    новое значение ссылки = S + A – P
- *S* – абсолютный адрес памяти, которому символ соответствует после перемещения
- *A* – дополнительное слагаемое (addend), хранимое непосредственно в байтах ссылки
- *P* – абсолютный адрес ссылки

```
snoop@earth:~/samples/2014$ objdump -s -j .data hello

hello:     file format elf32-i386

Contents of section .data:
 804a00c 00000000 00000000 e0840408            ............

snoop@earth:~/samples/2014$ objdump -d -M intel -s -j .text hello
...
080483e4 <main>:
 80483e4:       55                      push   ebp
 80483e5:       89 e5                   mov    ebp,esp
 80483e7:       83 e4 f0                and    esp,0xfffffff0
 80483ea:       83 ec 10                sub    esp,0x10
 80483ed:       c7 44 24 0c 00 00 00    mov    DWORD PTR [esp+0xc],0x0
 80483f4:       00
 80483f5:       e8 06 00 00 00          call   8048400 <func>
 80483fa:       8b 44 24 0c             mov    eax,DWORD PTR [esp+0xc]
 80483fe:       c9                      leave
 80483ff:       c3                      ret
```

Как изменились ссылки, попавшие в исполняемый код из файла `hello1.o`?