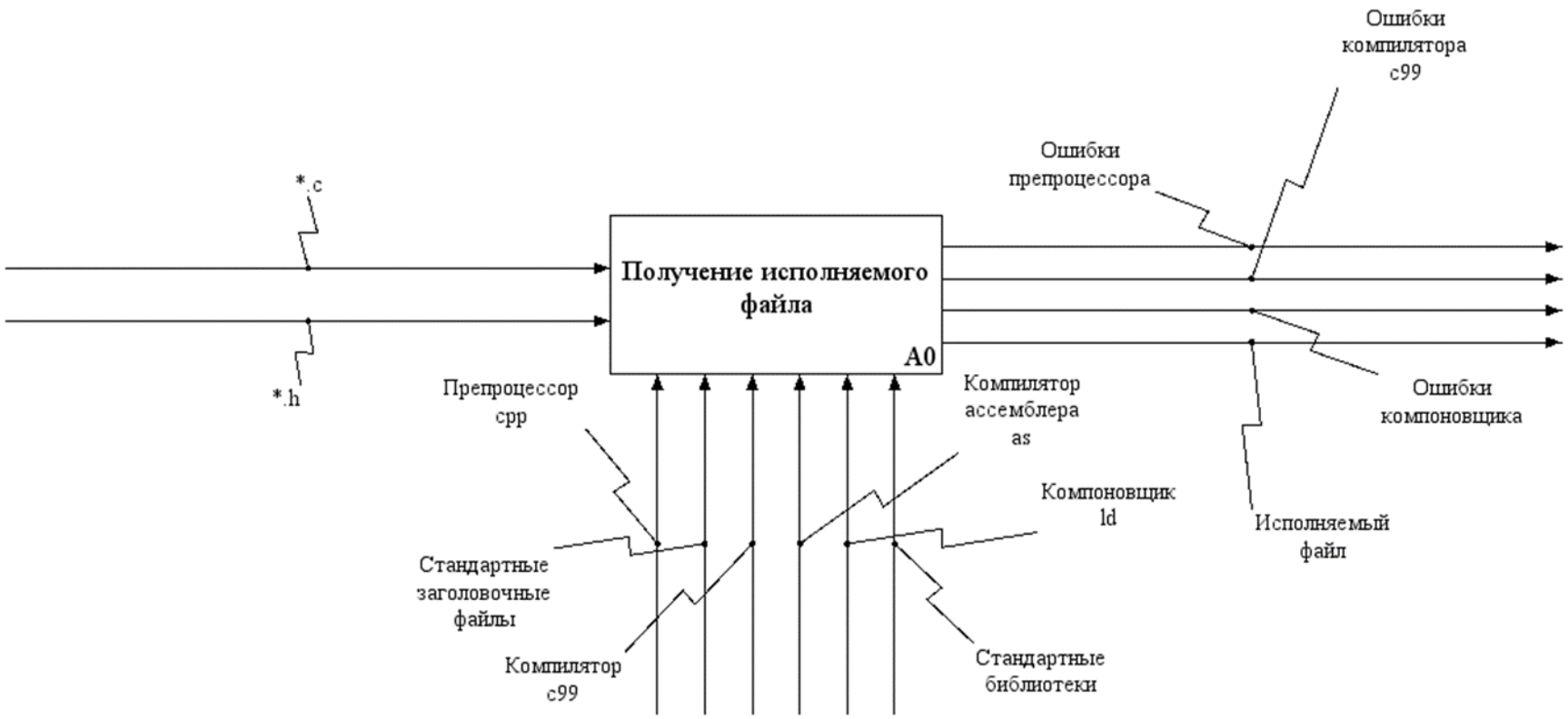
Получение исполняемого файла

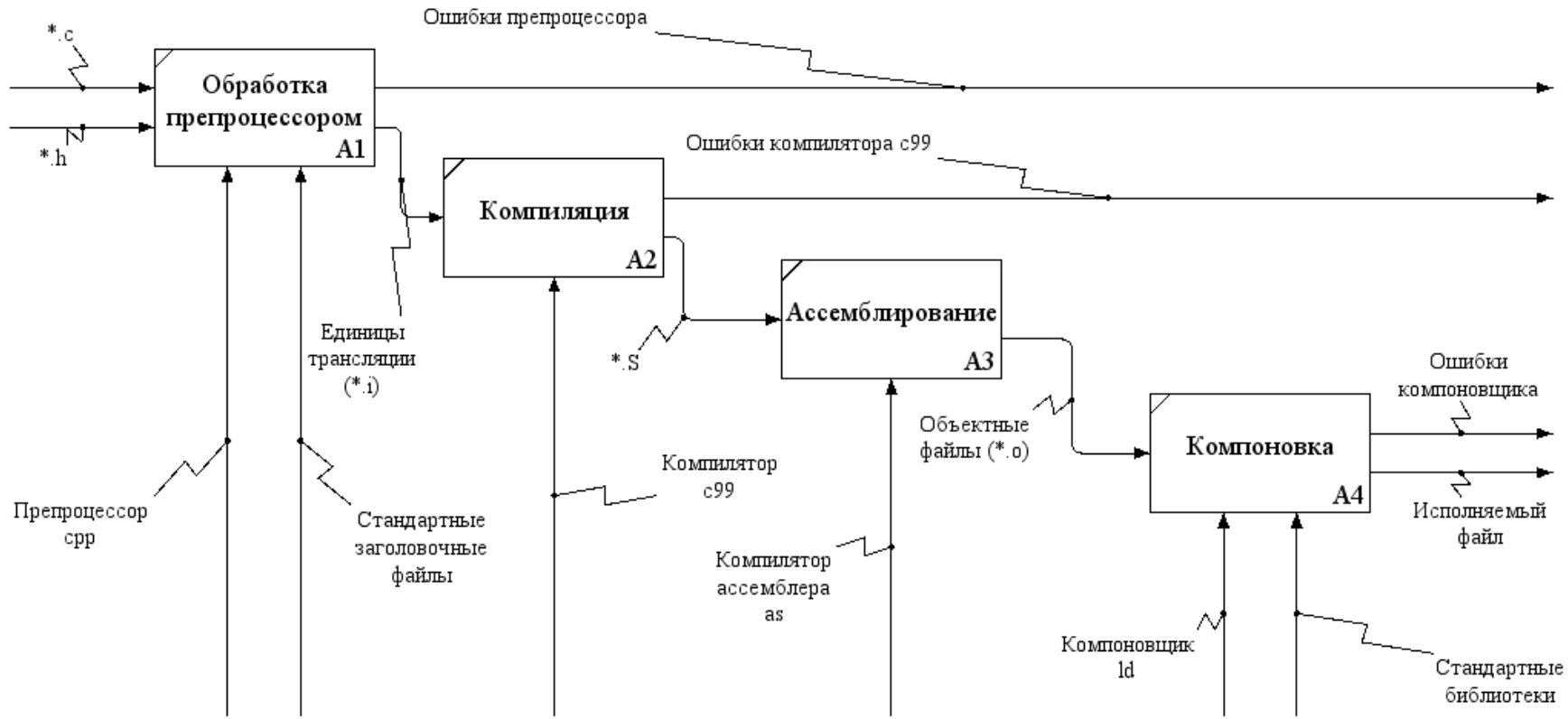
Простая программа на Си

```
1       /*
2       Простая программа на языке Си.
3       */
4
5       #include <stdio.h>
6
7       #define GREETING "Hello, world!"
8
9       int main(void)
10      {
11          puts(GREETING);
12
13          return 0;
14      }
```

Получение исполняемого файла (1)



Получение исполняемого файла (2)



A1: обработка препроцессором (1)

Препроцессор выполняет следующие действия:

- удаление комментариев;
- вставку файлов (директива include);
- текстовые замены (по-другому говорят - раскрытие макросов, директива define);
- условную компиляцию (директива if).

Файл, получаемый в результате работы препроцессора, называется *единицей трансляции*.

A1: обработка препроцессором (2)

```
cpp hello.c
cpp -o hello.i hello.c
cpp hello.c > hello.i
```

Результат работы препроцессора

```
...

extern int puts (const char *__s);

...

int main(void)
{
    puts("Hello, world!");

    return 0;
}
```

A2: трансляция на язык ассемблера (1)

Файл, полученный препроцессором, передается на вход транслятору c99, который переводит его с языка Си на язык ассемблера.

Язык ассемблера – это машинно-ориентированный язык низкого уровня. Команды языка ассемблера фактически один к одному соответствуют командам процессора.

Трансляция программы сначала на язык ассемблера позволяет:

- упростить реализацию и отладку транслятора;
- повысить его переносимость с одной платформы на другую.

A2: трансляция на язык ассемблера (2)

```
c99 -S -fverbose-asm -masm=intel hello.i
```

Результат работы компилятора

```
        .file  "hello.c"
// ...
        .text
        .section      .rodata
.LC0:
        .string       "Hello, world!"
        .text
        .globl main
        .type  main, @function
main:
.LFB0:
        .cfi_startproc
        endbr64
        push   rbp    #
```

```
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        mov    rbp, rsp      #,
        .cfi_def_cfa_register 6
# hello.c:11:     puts(GREETING);
        lea    rdi, .LC0[rip]       #,
        call   puts@PLT      #
# hello.c:13:     return 0;
        mov    eax, 0 # _3,
# hello.c:14: }
        pop    rbp    #
        .cfi_def_cfa 7, 8
        ret
//...
```

А3: ассемблирование в объектный файл

С языка ассемблера программа переводится в машинный код с помощью транслятора as.

```
as hello.s -o hello.o
```

На выходе этого транслятора получается не текстовый (как на двух предыдущих этапах), а двоичный файл. Этот файлы называется объектным файлом.

Объектный файл (1)

Объектный файл представляют собой блоки машинного кода и данных с неопределенными адресами ссылок на данные и подпрограммы в других объектных модулях, а также список своих подпрограмм и данных.

Объектный файл (2)

Объектный файл состоит из секций, которые содержат данные в широком смысле этого слова

- заголовки (метаинформация, необходимая для организации самого файла);
- код (.text);
- данные (.data, .rodata, .bss);
- таблицу символов (.symtab);
- …

Результат работы ассемблера

```
0000000000000000 <main>:
   0:  f3 0f 1e fa                endbr64
   4:  55                         push   rbp
   5:  48 89 e5                   mov    rbp,rsp
   8:  48 8d 3d 00 00 00 00       lea    rdi,[rip+0x0]          # f <main+0xf>
       b: R_X86_64_PC32    .rodata-0x4
   f:  e8 00 00 00 00             call   14 <main+0x14>      10:
R_X86_64_PLT32      puts-0x4
  14:  b8 00 00 00 00             mov    eax,0x0
  19:  5d                         pop    rbp
  1a:  c3                         ret
```

```
 U _GLOBAL_OFFSET_TABLE_
0000000000000000 T main
                 U puts
```

A4: компоновка (1)

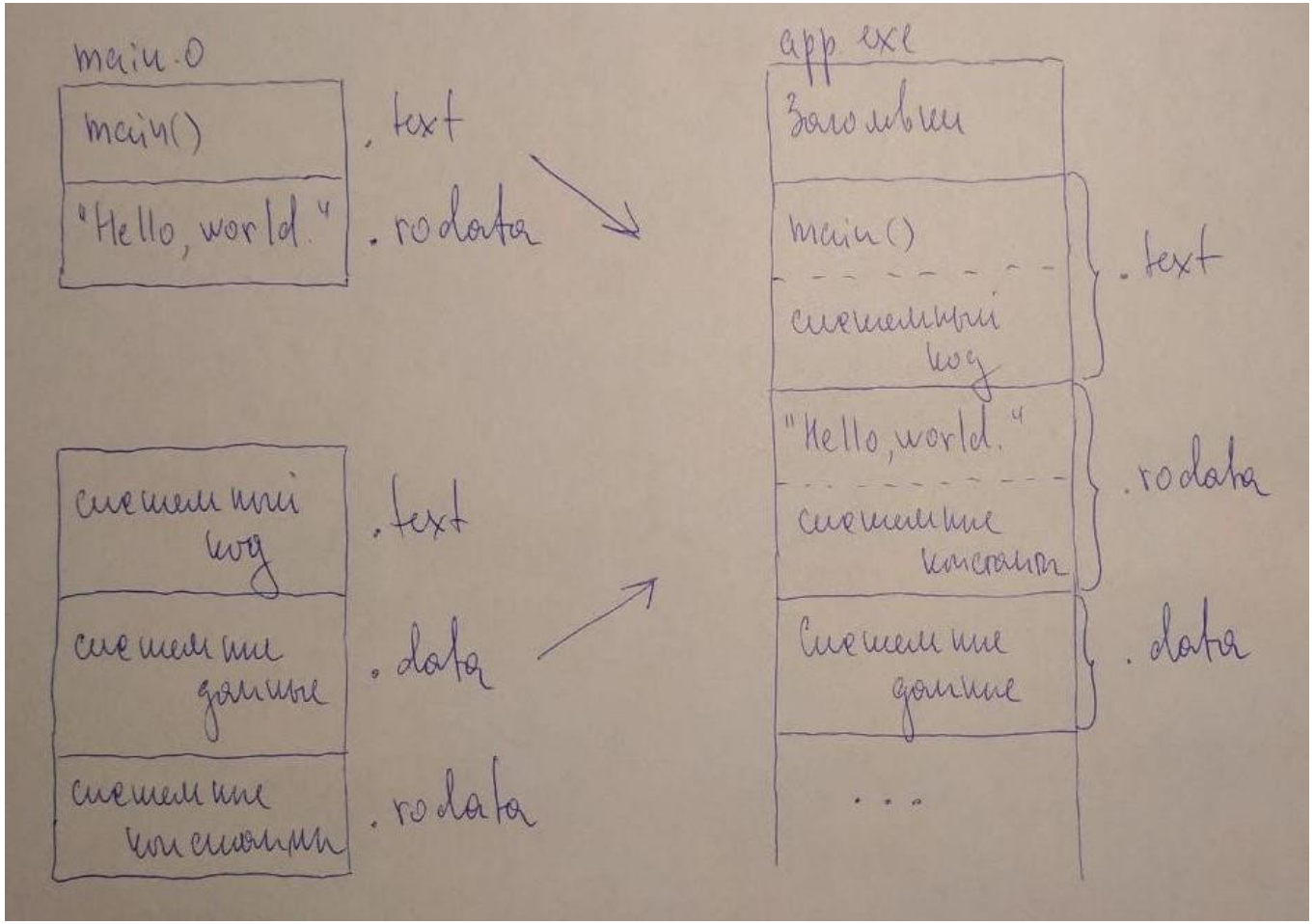
Чтобы получить исполняемый файл необходимо вызвать компоновщик.

```
ld другие_параметры -o hello.exe hello.o
```

В процессе получения исполняемого файла компоновщик решает несколько задач

- объединяет несколько объектных файлов в единый исполняемый файл;
- выполняет связывание переменных и функций, которые требуются очередному объектному файлу, но находятся где-то в другом месте;
- добавляет специальный код, который подготавливает окружение для вызова функции main, а после ее завершения выполняет обратные действия.

A4: компоновка (2)



Исполняемый файл

Исполняемый файл - файл, содержащий программу в виде, в котором она может быть (после загрузки в память и настройки по месту) исполнена компьютером.

Обычно исполняемый файл состоит из нескольких заголовков и нескольких секций.

- Заголовки содержат служебную информацию, описывающую различные свойства исполняемого файла и его структуру.
- Секции содержат данные в широком смысле этого слова (код, данные, служебная информация).

Назначение некоторых секций

.text	Содержит исполняемый код.
.bss	Содержит все неинициализированные статические и глобальные переменные.
.data	Содержит инициализированные глобальные и статические переменные, которые были проинициализированы во время компиляции.
.rodata	Содержит данные только для чтения, такие как литеральные строки (в том числе строки формата printf), константы, отладочную информацию.
Таблица импорта	Таблица импорта хранит пары имена функций и место, в которое загрузчик должен записать адрес этой функций.

Ключи компилятора (1)

Использовать четыре разные утилиты для получения объектного файла неудобно. Поэтому компилятор умеет выполнять все эти действия самостоятельно или с помощью вызова внешних утилит. Работа компилятора управляется ключами.

В большинстве POSIX-систем строка вызова компилятора выглядит следующим образом

```
имя_компилятора [ключи] [выходной_файл] файл_1 [файл_2]
```

Ключи компилятора (2)

-E	Компилятор выполняет только этап препроцессирования.
-S	Компилятор выполняет только трансляцию программы на язык ассемблера.
-c	Компилятор выполняет только получение объектного файла.
-o имя_файла	Задает имя выходного файла.
-std=XYZ	Задает стандарт языка Си, которые будет использовать при трансляции программы.
-Wall	Вынуждает компилятор выводить информацию о всех предупреждениях, с которыми он столкнулся во время компиляции.
-Werror	Вынуждает компилятор интерпретировать предупреждения.
-g[level]	Задает уровень отладочной информации, которая добавляется к объектному файлу.
-O[level]	Задает уровень оптимизации, которую выполняет компилятор.

Примеры запуска компилятора (1)

// 1. Препроцессирование

```
gcc -E main.c > main.i
```

// 2. Трансляция на язык ассемблера

```
gcc -std=c99 -Wall -Werror -S main.i
```

// 3. Ассемблирование

```
gcc -c main.s
```

// 4. Компоновка

```
gcc -o main.exe main.o
```

Примеры запуска компилятора (2)

// Вместо 1-3 можно написать

```
gcc -std=c99 -Wall -Werror -c main.c
```

// Вместо 1-4 можно написать

```
gcc -std=c99 -Wall -Werror -o main.exe main.c
```