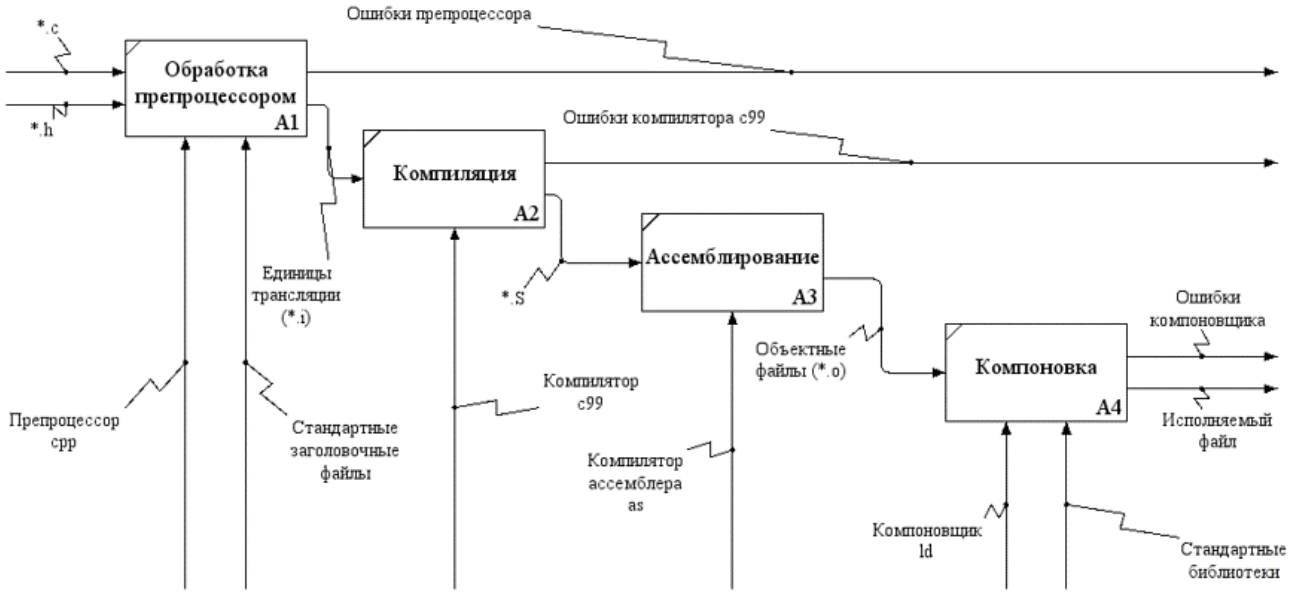
1. Этапы получение исполняемого файла

Рассмотрим простую программу на языке Си:

hello.c

```
 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. }
```

Процесс получения исполняемого файла из исходного кода состоит из четырех шагов, которые показаны на следующей IDEF0 диаграмме:



Каждый шаг выполняется с помощью отдельной утилиты и отличается от других шагов входными и выходными данными и решаемыми задачами. Рассмотрим эти шаги подробнее.

1.1. Препроцессирование

Препроцессор выполняет четыре действия:

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

Обычно результат своей работы препроцессор выводит непосредственно на экран. Его вывод можно перенаправить в файл с помощью перенаправления ввода/вывода или ключа "**-o имя_файла**".

```
// Запуск препроцессора
cpp hello.c > hello.i

// Вывод результатов работы на экран
cat hello.i

// Результаты (часть вывода пропущена)
...

int __attribute__((__cdecl__)) puts(const char *_Str);

...

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

    return 0;
}
```

Проанализируем результат работы препроцессора. В нашем случае препроцессор:

- удалил комментарий, который располагался в начале файле,
- выполнил включение заголовочного файла stdio.h (в примере оставлено только объявление функции puts, из-за которого это включение и требуется),
- заменил макрос GREETING на его значение.

Файл, который получен в результате работы препроцессора, называется *единицей трансляции*.

1.2. Трансляция на язык ассемблера

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

```
// Запуск транслятора (ключи запуска подробнее рассматриваются ниже)
c99 -S -fverbose-asm -masm=intel hello.i

// Вывод результатов работы на экран
cat hello.s

// Результаты
      .text
      .def  __main;     .scl  2;    .type 32;   .endef
      .section .rdata,"dr"
.LC0:
      .ascii "Hello, world!\0"
      .text
      .globl      main
      .def  main; .scl  2;    .type 32;   .endef
      .seh_proc   main
main:
      push  rbp    #
      .seh_pushreg      rbp
      mov   rbp, rsp     #,
      .seh_setframe     rbp, 0
```

```
        sub   rsp, 32      #,
        .seh_stackalloc   32
        .seh_endprologue
# main.c:10: {
        call   __main       #
# main.c:11:     puts(GREETING);
        lea   rcx, .LC0[rip]     #,
        call  puts   #
# main.c:13:     return 0;
        mov   eax, 0       # _3,
# main.c:14: }
        add   rsp, 32      #,
        pop   rbp    #
        ret
        .seh_endproc
        .ident      "GCC: (Rev1, Built by MSYS2 project) 8.2.1 20181214"
        .def  puts; .scl  2;    .type 32;   .endef
```

Благодаря ключу **"-fverbose-asm"** в файл помещаются дополнительные комментарии (строки, которые начинаются с символа '#'), которые позволяют сопоставить исходную программу на языке Си с полученной программой на языке ассемблере.

Язык ассемблера – это машинно-ориентированный язык низкого уровня. Команды языка ассемблера фактически один к одному соответствуют командам процессора. Они представляют собой более удобную символьную форму записи команд процессора и их аргументов.

Трансляция программы сначала на язык ассемблера позволяет:

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

1.3. Трансляция программы с языка ассемблера в машинный код

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 это двоичный файл, который содержит блоки машинного кода и метаданные. Метаданные описывают переменные и функции, которые требуются этому файлу (в нашем случае это функция puts) и которые этот файл может предоставить другим файлам (в нашем случае это функция main). Объектный файл не является самодостаточным и не может быть выполнен операционной системой.

Метаданные обычно образуют две таблицы:

- таблицу символов (symbol table), в которой перечисляются имена определенные (main) и упомянутые (puts) в исходном тексте программы;
- таблицу релокации (relocation table), в которой перечисляются «места» в объектном файле, которые нужно будет «поправить» при получении исполняемого файла.

Ниже представлены результаты «изучения» объектного файла с помощью специальных утилит.

```
// Запуск утилиты objdump (ключи запуска подробнее рассматриваются ниже)
objdump -drw -Mintel hello.o

// Результаты (часть вывода опущена)
...
0000000000000000 <main>:
   0:   55                      push   rbp
   1:   48 89 e5                mov    rbp,rsp
   4:   48 83 ec 20             sub    rsp,0x20
   8:   e8 00 00 00 00          call   d <main+0xd>     9: R_X86_64_PC32
__main
   d:   48 8d 0d 00 00 00 00    lea    rcx,[rip+0x0]        # 14 <main+0x14>
10: R_X86_64_PC32       .rdata
  14:   e8 00 00 00 00          call   19 <main+0x19>   15: R_X86_64_PC32
puts
  19:   b8 00 00 00 00          mov    eax,0x0
  1e:   48 83 c4 20             add    rsp,0x20
  22:   5d                      pop    rbp
  23:   c3                      ret
  24:   90                      nop
...
```

Ключ "**-d**" отвечает за дизассемблирование (т.е. обратное преобразование из машинного кода на язык ассемблера) объектного файла. В этой части вывода утилиты objdump удобно сопоставить команды языка ассемблера (они в правой части) и команды процессора (они в левой части).

Ключ "**-r**" отвечает за вывод «мест», которые нужно будет поправить компоновщику.

```
// Вывод таблицы релокации
objdump -r hello.o

// Результаты (часть вывода опущена)
...
RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000009 R_X86_64_PC32     __main
0000000000000010 R_X86_64_PC32     .rdata
0000000000000015 R_X86_64_PC32     puts
...
```

Если вы внимательно посмотрите на результат дизассемблирования и таблицу релокации, то увидите связь между ними.

```
// Вывод таблицы символов с помощью утилиты nm
nm hello.o

// Результаты
0000000000000000 b .bss
0000000000000000 d .data
0000000000000000 p .pdata
0000000000000000 r .rdata
0000000000000000 r .rdata$zzz
0000000000000000 t .text
0000000000000000 r .xdata
                 U __main
0000000000000000 T main
                 U puts
```

1.4. Компоновка

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 (в стандартной библиотеке или другом объектном файле);
- добавляет специальный код, который подготавливает окружение для вызова функции main, а после ее завершения выполняет обратные действия.

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) и представляют собой непрерывные области адресного пространства со своими атрибутами доступа.

Назначение некоторых секций

Название	**Назначение**
.text	Содержит исполняемый код.
.bss	Содержит все неинициализированные статические и глобальные переменные. Эта секция не занимает места в объектном файле, это лишь «заполнитель».
.data	Содержит инициализированные глобальные и статические переменные, которые были проинициализированы во время компиляции.
.rodate	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.

2. Ключи компилятора

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 ключом. В большинстве POSIX-систем строка вызова компилятора выглядит следующим образом

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

В нашем случае **`имя_компилятора`** – это gcc. Рассмотрим назначение основных ключей этого компилятора.

Ключ	**Описание**
`-E`	Компилятор выполняет только этап препроцессирования.
`-S`	Компилятор выполняет только трансляцию программы на язык ассемблера.

-c	Компилятор выполняет только получение объектного файла (т.е. будут выполнены три первых этапа получения исполняемого файла).
-o имя_файла	Задает имя выходного файла.
-std=XYZ	Задает стандарт языка Си, которые будет использовать при трансляции программы. В нашем случае -std=c99.
-Wall	Вынуждает компилятор выводить информацию о всех предупреждениях, с которыми он столкнулся во время компиляции. Наличие предупреждений не мешает получению исполняемого файла. Однако с их помощью компилятор пытается обратить внимание программиста на нестандартное использование той или иной конструкции языка, которое может привести к возникновению ошибки.
-Werror	Вынуждает компилятор интерпретировать предупреждения, которые выдаются в процессе компиляции, как ошибки.
-pedantic	Вынуждает компилятор педантично следовать указанному стандарту.
-g[level]	Задает уровень отладочной информации, которая добавляется к объектному файлу. level – это число, которое принимает значения от 0 (отсутствие отладочной информации) до 3 (максимальное количество отладочной информации).
-O[level]	Задает уровень оптимизации, которую выполняет компилятор. level – это число, которое принимает значения от 0 (отсутствие оптимизации) до 3 (максимальный уровень оптимизации).

Обязательными для использования при сборке лабораторных работ являются ключи **"-std=c99 -Wall -Werror"**.

Рассмотрим примеры вызова компилятора

```
// 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

// 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
```