Сборка многомодульных программ

1. Понятие единицы трансляции, области видимости.
    1. *Идентификатор* как имя объекта (переменной) или функции.
    2. *Область видимости* имени: файл, блок (составной оператор), функция, прототип функции. Начало и конец области видимости.
    3. *Единица трансляции* — один файл с исходным кодом (на Си или языке ассемблера).
2. Понятие связывания.
    1. *Связывание* имён:
        - *внешнее связывание* — имя доступно в нескольких единицах трансляции;
        - *внутреннее связывание* — имя доступно только в своей единице трансляции;
        - *без связывания* — для имён, не описывающих объекты (переменные) или функции, например, теги структур, параметры функций, локальные переменные без ключевого слова `extern`.
    2. Связывание по умолчанию для переменных (кроме локальных) и функций в Си — внешнее. Для внутреннего связывания используется ключевое слово `static`.
    3. В языке ассемблера наоборот: локальные и нелокальные метки соответствуют внутреннему связыванию. Для внешнего связывания используется директива `global`.
    4. Для объявления использования внешнего имени используется директива `extern`.
    5. Пример 26-1: внешнее связывание функций в языке Си.
    6. Пример 26-2: внешнее связывание функций в языке ассемблера.
    7. Можно использовать в единице трансляции на языке ассемблера внешние имена из единицы трансляции на Си и наоборот.
3. Понятие объектного модуля. Процесс сборки программы.
    1. Процесс сборки программы:
        - *препроцессирование* — подстановка текстов заголовочных файлов (директива `#include`), условная трансляция, макроподстановки;
        - *компиляция* — получение из каждой единицы трансляции кода на машинном языке и генерация *объектного модуля*;
        - *линковка* — объединение объектных модулей вместе с библиотеками в исполняемый файл.
    2. *Объектный модуль* — представленные в машинном виде объекты и функции, содержащиеся в единице трансляции, вместе с *таблицей имён*. В таблицу имён попадают только те объекты и функции, которые имеют внешнее связывание.
    3. Разрешение связей во время линковки: проверяется, что для каждого заявленного внешнего имени есть ровно одно определение в данном наборе объектных модулей. Невыполнение этого правила приводит к ошибке линковки.
    4. Препроцессирование и компиляция, как правило, происходят одновременно. Линковка является отдельным этапом для многомодульных программ. В случае одномодульной программы линковка может быть выполнена одновременно с первыми двумя этапами.
    5. Пример 26-3: компиляция и линковка Си-программы.
    6. Пример 26-4: компиляция и линковка ассемблерной программы.
    7. При использовании gcc для линковки, если не указано обратное, будет подключаться стандартная библиотека языка Си. Выполнение будет начинаться с функции `main` с учётом следующего замечания.
    8. Замечание для Windows: все внешние имена при использовании `cdecl` снабжаются ведущим подчёркиванием (то есть `main` превращается в `_main`). Компилятор языка Си делает это автоматически, а вот в программе на языке

ассемблера это необходимо делать вручную. Макросы из библиотеки `io.inc` позволяли раньше не думать об этом:

- конструкция `CEXTERN` *имя* в UNIX преобразовывалась в extern *имя*, а в Windows — в extern *_имя*;
- конструкция `CMAIN`, аналогично, преобразовывалась либо в `main`, либо в `_main`.

4. Понятие библиотеки как совокупности объектных модулей. Подключение библиотек.
    1. Библиотека имеет имя, по которому её можно подключить.
    2. Линковщик, входящий в состав gcc, если не указано обратное, подключает стандартную библиотеку языка Си `libc`.
    3. Математическая библиотека языка Си называется `libm`.
    4. Пример 26-5: линковка с использованием библиотек.
5. Задачи 26-1, 26-2.
6. Заголовочные файлы.
    1. В Си-программах необходимо для каждого внешнего имени обеспечить его объявление. Поскольку таких внешних имён может быть много, чтобы не вписывать их объявления вручную в каждую единицу трансляции (файлы `.c`), используются заголовочные файлы с расширением `.h`.
    2. Препроцессор, встречая директиву `#include`, подставляет текст заголовочного файла полностью в данный файл.
    3. Системные заголовочные файлы должны использовать угловые скобки, а локальные файлы — двойные кавычки.
    4. Разница между подключением заголовка и линковкой с библиотекой: в первом случае получаем только объявления, но не сами объекты. Во втором — наоборот. Итого: необходимо делать и то, и другое.
    5. Пример 26-6: использование заголовочных файлов.
7. Задача 26-3.

Примеры

Пример 26-1

Внешнее связывание функций в языке Си

Файл `file1.c`:

```
extern int
fib(int);

static int
fib5(void)
{
    return fib(5);
}
```

Файл `file2.c`:

```
int
fib(int n)
{
    return n > 2 ? fib(n - 1) + fib(n - 2) : 1;
}
```

Пример 26-2

Внешнее связывание функций в языке ассемблера

Файл `file1.asm`:

```
EXTERN fib

fib5:
    PUSH    EBP
    MOV     EBP, ESP

    SUB     ESP, 8

    MOV     DWORD [ESP], 5
    CALL    fib

    LEAVE
    RET
```

Файл `file2.asm`:

```
GLOBAL fib
fib:
    PUSH    EBP
    MOV     EBP, ESP

    SUB     ESP, 8

    MOV     EAX, DWORD [EBP + 8]
    CMP     EAX, 2
    JG      .1

    MOV     EAX, 1
    JMP     .2

.1:
    MOV     DWORD [ESP], EAX
    DEC     DWORD [ESP]
    CALL    fib

    MOV     DWORD [ESP + 4], EAX
    DEC     DWORD [ESP]
    CALL    fib

    ADD     EAX, DWORD [ESP + 4]

.2:
    LEAVE
    RET
```

Пример 26-3

Компиляция и линковка Си-программы.

Команда для компиляции Си-модуля с использованием gcc:

`gcc [ `*`дополнительные-опции`*` ] -c -o `*`объектный-модуль.o единица-трансляции.c`

Ключ -c: компилировать исходные файлы, но не линковать. Стадия линковки просто не выполняется. Конечный вывод происходит в форме объектного файла для каждого исходного файла. По умолчанию, имя объектного файла делается из имени исходного файла заменой суффикса '.c' на '.o'.

Ключ -std=c99 позволяет компилировать код программы на Си в соответствии со стандартом C99.

Ключ -m32 позволяет компилировать и собирать 32-битные программы на 64-битной системе.

Ключ -o *имя_файла*: поместить вывод в файл с именем *имя_файла*. Эта опция применяется вне зависимости от вида порождаемого файла, есть ли это исполняемый файл, объектный файл или файл с исходным кодом.

Команда для линковки нескольких модулей с использованием gcc:

```
gcc [ дополнительные-опции ] -o исполняемый-файл объектный-модуль1 объектный-модуль2 ...
```

Традиционно на UNIX-системах расширение для объектного модуля — ".o", а исполняемые файлы не имеют расширения.

Пример сборки программы из двух модулей:

```
gcc -std=c99 -m32 -c -o file1.o file1.c   компиляция первого модуля, получаем file1.o
gcc -std=c99 -m32 -c -o file2.o file2.c   компиляция второго модуля, получаем file2.o
gcc -o program file1.o file2.o       линковка file1.o и file2.o, получаем исполняемый файл program
```

Пример 26-4

Компиляция и линковка ассемблерной программы.

Команда для компиляции ассемблерного модуля с использованием nasm:

```
nasm [ дополнительные-опции ] -f elf32 -o объектный-модуль.o единица-трансляции.asm ; на UNIX-системах
nasm [ дополнительные-опции ] -f win32 -o объектный-модуль.o единица-трансляции.asm ; на Windows
```

Линковка может быть осуществлена с использованием gcc так же, как и в примере 26-3.

Пример сборки программы из двух модулей на UNIX:

```
nasm -f elf32 -o file1.o file1.asm     компиляция первого модуля, получаем file1.o
nasm -f elf32 -o file2.o file2.asm     компиляция второго модуля, получаем file2.o
gcc -m32 -o program file1.o file2.o    линковка file1.o и file2.o, получаем исполняемый файл program
```

Пример 26-5

Для линковки с дополнительными библиотеками используются опции `-l` и `-L`. Опция `-l` используется для указания имени библиотеки, с которой будет производиться линковка. При этом префикс `lib` отбрасывается.

Пример линковки с библиотекой `libm`:

```
gcc -m32 -lm -o program file1.o file2.o
```

Если библиотека расположена в нестандартном месте (стандартное место зависит от платформы), то необходимо указать путь к ней опцией `-L`.

Пример линковки с указанием пути к библиотекам:

```
gcc -m32 -L/usr/local/lib -lm -o program file1.o file2.o
```

Пример 26-6

Вынесем объявление функции `fib()` из примеров 1 и 2 в отдельный заголовочный файл.

Файл `fib.h` с объявлением:

```
int
fib(int number);
```

Файл `main.c` с использованием функции `fib()`:

```
#include <stdio.h>

#include "fib.h"

int
main(void)
{
    printf("fib(5) = %d\n", fib(5));
    return 0;
}
```

Файл `fib.c` с реализацией функции `fib()`:

```
#include "fib.h"

int
fib(int number)
{
    return number > 2 ? fib(number - 1) + fib(number - 2) : 1;
}
```

Вопросы к примеру:

1. Зачем в файле `fib.c` подключается заголовок?
2. Можно ли пометить функцию `fib` как `extern`? Как `static`?
3. Какие команды для сборки необходимо использовать?

Комментарий: может понадобиться добавить к команде компиляции ключ `-I.` для того, чтобы компилятор нашёл локальный заголовочный файл.

Задачи

Задача 26-1

Программа `quux` состоит из одного Си-модуля в файле `foo.c` и одного ассемблерного модуля в файле `bar.asm`. Выписать команды для сборки этой программы. Программа должна быть слинкована с математической библиотекой языка Си.

Решение:

```
gcc -m32 -std=c99 -c -o foo.o foo.c
nasm -f elf32 -o bar.o bar.asm
gcc -m32 -lm -o quux foo.o bar.o
```

Задача 26-2

Написать на языке Ассемблера программу, вводящую два целых 32-разрядных числа, и выводящая максимальное из них. Запрещается использовать библиотеку макросов `io.inc`. Считать, что программа будет работать в UNIX-окружении. Выписать команды для сборки программы.

Программа в файле `max.asm`:

```
SECTION .rodata

    fmtS    DB "%d%d", 0
    fmtP    DB "%d", 10, 0

SECTION .text

EXTERN scanf
EXTERN printf

GLOBAL main
main:
    LEA     ECX, [ESP + 4]              ; выравниваем стек
    AND     ESP, -16
    PUSH    DWORD [ECX - 8]

    PUSH    EBP                         ; создаём фрейм
    MOV     EBP, ESP

    PUSH    ECX                         ; сохраняем ECX
    SUB     ESP, 20                     ; резервируем два параграфа по 16 байтов

    MOV     DWORD [ESP], fmtS           ; готовим параметры для scanf()
    LEA     EAX, [ESP + 12]             ; локальная переменная
    MOV     DWORD [ESP + 4], EAX
    LEA     EAX, [ESP + 16]             ; локальная переменная
    MOV     DWORD [ESP + 8], EAX

    CALL    scanf

    MOV     EAX, DWORD [ESP + 16]       ; вычисляем максимум
    CMP     EAX, DWORD [ESP + 12]
    CMOVL   EAX, DWORD [ESP + 12]

    MOV     DWORD [ESP], fmtP           ; готовим параметры для printf()
```

```
    MOV     DWORD [ESP + 4], EAX

    CALL    printf

    ADD     ESP, 20                     ; восстанавливаем стек
    POP     ECX
    LEA     ESP, [ECX - 4]

    XOR     EAX, EAX                    ; возвращаем 0
    RET
```

Команды для сборки:

```
nasm -f elf32 -o max.o max.asm
gcc -m32 -o max max.o
```

Дополнительные вопросы:

1. Каким связыванием обладают `fmtS`, `fmtP`, `main`, `scanf`, `printf`?
2. Какие изменения необходимо внести в программу и процесс сборки, чтобы собирать программу под Windows?

Задача 26-3

Написать на языке ассемблера функцию `countof()`, считающую количество вхождений символа в строку, ограниченную `\0`. Функция должна использовать соглашение `cdecl` и иметь следующий прототип.

```
int
countof(const char *s, char c);
```

Реализовать на языке Си функцию `main()`, считывающую две строки без пробелов, не более 80 символов каждая, и выводящую ту из них, которая имеет больше вхождений символа `$`.

Указать команды, необходимые для сборки программы. Вынести объявление функции в заголовчный файл.

Решение: файл `countof.h`.

```
int
countof(const char *s, char c);
```

Решение: файл `countof.asm`.

```
SECTION .text

GLOBAL countof
countof:
    PUSH    EBP                         ; стандартный пролог
    MOV     EBP, ESP

    XOR     EAX, EAX                    ; счётчик
    MOV     CL, BYTE [EBP + 12]         ; символ в CL
    MOV     EDX, DWORD [EBP + 8]        ; адрес очередного символа строки в EDX

.1:
    CMP     BYTE [EDX], 0               ; проверяем на конец строки
```

```
    JE      .3

    CMP     BYTE [EDX], CL                  ; проверяем на совпадение
    JNE     .2

    INC     EAX                             ; совпало, увеличиваем счётчик

.2:
    INC     EDX                             ; идём дальше по строке
    JMP     .1

.3:
    LEAVE                                   ; стандартный эпилог
    RET
```

Решение: файл `main.c`.

```
#include <stdio.h>

#include "countof.h"

int
main(void)
{
    char a[81], b[81];
    int na, nb;

    scanf("%s %s", a, b);

    na = countof(a, '$'); nb = countof(b, '$');

    printf("%s\n", na > nb ? a : b);
    return 0;
}
```

Решение: последовательность команд для сборки.

```
gcc -m32 -std=c99 -c -o main.o main.c
nasm -f elf32 -o countof.o countof.asm
gcc -m32 -o program main.o countof.o
```

Утилита make

make — утилита, автоматизирующая процесс преобразования файлов из одной формы в другую. Чаще всего это компиляция исходного кода в объектные файлы и последующая компоновка в исполняемые файлы или библиотеки. Утилита использует специальные make-файлы, в которых указаны зависимости файлов друг от друга и правила для их удовлетворения. На основе информации о времени последнего изменения каждого файла make определяет и запускает необходимые программы.

Использование: make [-f make-файл] [цель] ...

Если опция -f не указана, используется имя по умолчанию для make-файла — Makefile (без расширения).

make открывает make-файл, считывает правила и выполняет команды, необходимые для создания указанной цели.

Стандартные цели для сборки:
all — собрать программу (получить исполняемый файл)
clean — очистить дистрибутив (удалить из дистрибутива объектные и исполняемые файлы, созданные в процессе компиляции)

Make-файл

Программа make выполняет команды согласно правилам, указанным в специальном файле. Этот файл называется make-файл (makefile, мейкфайл). Как правило, make-файл описывает, каким образом нужно компилировать и компоновать программу.

make-файл состоит из правил и переменных. Правила имеют следующий синтаксис:
```
цель1 цель2 ...: реквизит1 реквизит2 ...
    команда1
    команда2
    ...
```

Правило представляет собой набор команд, выполнение которых приведёт к сборке файлов-целей из файлов-реквизитов.

Правило сообщает make, что файлы, получаемые в результате работы команд (цели) являются зависимыми от соответствующих файлов-реквизитов. make никак не проверяет и не использует содержимое файлов-реквизитов, однако, указание списка файлов-реквизитов требуется только для того, чтобы make убедилась в наличии этих файлов перед началом выполнения команд и для отслеживания зависимостей между файлами.

Обычно цель представляет собой имя файла, который генерируется в результате работы указанных команд. Целью также может служить название некоторого действия, которое будет выполнено в результате выполнения команд (например, цель clean в make-файлах для компиляции программ обычно удаляет объектные файлы, созданные в процессе компиляции).

Строки, в которых записаны команды, должны начинаться с символа табуляции.

Рассмотрим несложную программу на Си. Пусть программа program состоит из пары файлов кода — main.c и lib.c, а также из одного заголовочного файла — defines.h, который подключён в оба файла кода. Поэтому, для создания program необходимо из пар (main.c defines.h) и (lib.c defines.h) создать объектные файлы main.o и lib.o, а затем скомпоновать их в program. При сборке вручную требуется дать следующие команды:
```
gcc -m32 -std=c99 -c -o main.o main.c
gcc -m32 -std=c99 -c -o lib.o lib.c
gcc -m32 -o program main.o lib.o
```

Если в процессе разработки программы в файл defines.h будут внесены изменения, потребуется перекомпиляция обоих файлов и линковка, а если изменяется lib.c, то повторную компиляцию main.o можно не выполнять.

Таким образом, для каждого файла, который мы должны получить в процессе компиляции, нужно указать, на основе каких файлов и с помощью какой команды он создаётся. Программа make на основе этих данных выполняет следующее: собирает из этой информации правильную последовательность команд для получения требуемых результирующих файлов; инициирует создание требуемого файла только в случае, если такого файла не существует, или он старше, чем файлы, от которых он зависит.

Если при запуске make явно не указать цель, то будет обрабатываться первая цель в make-файле, имя которой не начинается с символа «.».

Если не нужны дополнительные ключи -m32 (когда программа уже выполняется на 32-битной системе) и std=c99, то для программы program достаточно написать следующий make-файл:

```
program: main.o lib.o
	gcc -o program main.o lib.o
main.o lib.o: defines.h
```

В имени второй цели указаны два файла и для этой же цели не указана команда компиляции. Кроме того, нигде явно не указана зависимость объектных файлов от «*.c»-файлов. Дело в том, что программа make имеет предопределённые правила для получения файлов с определёнными расширениями. Так, для цели - объектного файла (расширение «.o») при обнаружении соответствующего файла с расширением «.c» будет вызван компилятор «gcc -c» с указанием в параметрах этого «.c»-файла и всех файлов-зависимостей.

Предположим, что к проекту добавился второй заголовочный файл lib.h, который включается только в lib.c. Тогда make-файл увеличится ещё на одну строчку:

```
lib.o: lib.h
```

Таким образом, один целевой файл может указываться в нескольких целях. При этом полный список зависимостей для файла будет составлен из списков зависимостей всех целей, в которых он участвует, создание файла будет производиться только один раз.

Цель clean для удаления объектных файлов (любых файлов с расширением .o)

```
clean:
	rm *.o
```

Директивы условной компиляции

Имеется несколько директив, которые дают возможность выборочно компилировать части исходного кода программы. Этот процесс называется условной компиляцией и широко используется фирмами, которые поставляют и поддерживают многие специальные версии одной программы.

Директивы #if, #else, #elif и #endif

Возможно, самыми распространенными директивами условной компиляции являются #if, #else, #elif и #endif. Они дают возможность в зависимости от значения константного выражения включать или исключать те или иные части кода.

В общем виде директива #if выглядит таким образом:

```
#if константное выражение
   последовательность операторов
#endif
```

Если находящееся за #if константное выражение истинно, то компилируется код, который находится между этим выражением и #endif. В противном случае этот промежуточный код пропускается. Директива #endif обозначает конец блока #if.

Например, в следующем фрагменте для определения знака денежной единицы используется значение ACTIVE_COUNTRY (для какой страны):

```
#define US 0
#define ENGLAND 1
#define FRANCE 2

#define ACTIVE_COUNTRY US

#if ACTIVE_COUNTRY == US
  char currency[] = "dollar";
#elif ACTIVE_COUNTRY == ENGLAND
  char currency[] = "pound";
#else
  char currency[] = "franc";
#endif
```

Директивы #ifdef и #ifndef

Другой способ условной компиляции — это использование директив #ifdef и #ifndef, которые, соответственно, означают "if defined" (если определено) и "if not defined" (если не определено). В общем виде #ifdef выглядит таким образом:

```
#ifdef имя_макроса
      последовательность операторов
#endif
```

Блок кода будет компилироваться, если имя макроса было определено ранее в операторе #define.

В общем виде оператор #ifndef выглядит таким образом:

```
#ifndef имя_макроса
      последовательность операторов
#endif
```

Блок кода будет компилироваться, если имя макроса еще не определено в операторе #define.

И в #ifdef, и в #ifndef можно использовать оператор #else или #elif. Например,

```
#include <stdio.h>

#define TED 10

int main(void)
{
   #ifdef TED
      printf("Привет, Тед\n");
   #else
      printf("Привет, кто-нибудь\n");
   #endif
   #ifndef RALPH
      printf("А RALPH не определен, Ральфу не повезло.\n");
   #endif
   return 0;
}
```

Вывод:
Привет, Тед
А RALPH не определен, Ральфу не повезло.