Лабораторное занятие №2. Язык Си. Библиотеки. Сборка

Темы

На занятии необходимо рассмотреть дополнительные опции компилятора *gcc*; познакомиться с библиотеками: подключение стандартных библиотек, создание и подключение статических библиотек, работа с динамически подключаемыми библиотеками; рассмотреть основные возможности системы сборки программ *make*.

Ход занятия

Литература

Обсудить литературу по языку программирования *Си*:

Русскоязычная:

Брайан Керниган, Деннис Ритчи. Язык программирования С. — Москва: «Вильямс», 2015. — 304 с.

и т.д.

Англоязычная:

German Gonzalez-Morris, Ivor Horton. Beginning C. From Beginner to Pro. 6-th edition - Apress, 2020, 674 p.

и т.д.

Задача №1

Получите от пользователя значение переменной x и вычислите значения функций $f(x)=\exp(-|x|)\sin(x)$ и $g(x)=\exp(-|x|)\cos(x)$.

Напишите две версии программы. *Версия 1:* все функции размещены в одном файле с исходным кодом и *версия 2:* функции разнесены по нескольким файлам (см. приложение в конце).

Замечание: на примере однофайловой программы познакомить с основными этапами обработки исходного файла компилятором *gcc* и основными опциями для остановки на выбранном этапе, а также некоторыми дополнительными полезнами опциями. Можно в консоли создать файлы с соответствующими командами и сделать их исполняемыми только для себя.

Этап 1. *Обработка препроцессором*

```
gcc -E task1.c > task1.i                gcc -E -o task1.i task1.c
ls
less task1.i
```

Этап 2. *Компиляция в ассемблерный код*

```
gcc -S task1.c                          gcc -S task1.i
ls
less task1.s
```

Этап 3. *Ассемблирование — трансляция в машинный код*

```
gcc -c task1.c                          gcc -c task1.s
ls
```

Этап 4. *Линковка* (*связывание*, *редактирование связей*)
```
gcc -o task1 task1.o -lm
ls
```

Дополнительные опции
Указание стандарта: `-std=c99` `c90`
Опции оптимизации: `-O1` `-O2` `-O3`
Подготовка к отладке: `-g`
Вывод предупреждений: `-Wall` (можно еще о `-Werror`)
Путь к заголовочным файлам: `-Ipath/to/directory/with/header/files`
Путь к библиотеке (может в раздел библиотек): `-Lpath/to/directory/with/library`
Тип файла: `-x c` (`f95` `java` `c++`)

Созадние библиотек пользователя

На примере **Задачи №1** рассмотреть назначение и преимущества библиотек пользователя.

Статические библиотеки (static)

Сначала создаем объектные файлы всех файлов с исходными текстами, которые нужно включить в библиотеку
```
gcc -c functs.c
```
Создаем библиотеку — архивный файл `libfun.a`, созданный утилитой `ar`
```
ar r libfun.a *.o
```
Создаем указатель для библиотеки, чтобы компоновщик мог находить в ней функции. Используем `ranlib`
```
ranlib libfun.a
```
Оба шага можно совместить с помощью опции `s` утилиты `ar`
```
ar rs libfun.a functs.o
```

Когда есть библиотека и заголовочныйфайл, содержащий входящие в нее функции, их можно использовать в программах.
```
gcc -I./include -L./lib -o task2 task2.c -lfun -lm
```
Если в указанной директории есть файлы и статических библиотек и библиотек совместного доступа с одним именем, то к программе будет подключена именно библиотека совместного доступа. Если нужно подключить именно статическую библиотеку нужно использовать опцию `-static` утилиты `gcc`. Или вместо этого ее нужно точно указать в строке вызова, как объектный файл без опции `-l`.

Совместного доступа (shared)

Сначала создаем объектные файлы всех файлов с исходными текстами, которые нужно включить в библиотеку в режиме генерации независимого от положения кода. Для этого используются опции или `-fpic` или `-FPIC`. Предпочтительнее использовать первый ключ. Но если таблица перемещаемого кода слишком мала, то с первой опцией компилятор выдает сообщение об ошибке и нужно использовать вторую.
```
gcc -c -fpic functs.c
```
Создаем совместно используемую библиотеку
```
gcc -shared -o libfun.so functs.o
```
Создавать указатель, как в случае статических библиотек, не требуется. Используется так же, как и статическая. Команда компилятора та же

```
gcc -I./include -L./lib -o task2 task2.c -lfun -lm
```

Полезной утилитой в случае библиотек совместного доступа является `ldd` — с ее помощью можно узнать, какие библиотеки совместного доступа использует программа

```
ldd task2
     linux-gate.so.1 =>  (0x00934000)
     libfun.so => not found
     libm.so.6 => /lib/i386-linux-gnu/libm.so.6 (0x00335000)
     libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0x00b8e000)
     /lib/ld-linux.so.2 (0x00533000)
```

Аналогично можно посмотреть зависимости библиотеки совместного доступа

```
ldd libfun.so
     linux-gate.so.1 =>  (0x00828000)
     libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0x0038a000)
     /lib/ld-linux.so.2 (0x00117000)
```

При запуске проблема нет система не находит совместно используемую библиотеку — система ищет библиотеки по умолчанию только в каталогах `/lib` и `/usr/lib`.

Первое решение: при линковке программу указать место, где система будет искать совместно используемые библиотеки. Для этого есть опция (без пробелов после запятой) `-Wl,-rpath`

```
gcc   -I./include   -L./lib   -o   task2   task2.c   -lfun   -lm
-Wl,-rpath,./lib
```

Теперь можно спокойно запускать программу, т. к. все совместно используемiе библиотеки система будет искать в каталоге `lib` текущего.

Второе решение: использовать переменную окружения `LD_LIBRARY_PATH`. В этой переменной хранится разделенный вертикальными двоеточиями список катологов, в которых система ищет совместно используемые библиотеки пред просмотром стандартных каталогов. Если эта переменная задана, то линкер при сборке программы будет просматривать заданные в ней каталоги в дополнение тем, которые заданы с помощью опции `-L`.

```
./task2
./task2: error while loading shared libraries: libfun.so: cannot
open shared object file: No such file or directory
export LD_LIBRARY_PATH=./lib
echo $LD_LIBRARY_PATH
./lib
./task2
x: 3.5
f(3.500000) = -0.0105927      g(3.500000) = -0.0282785
```

Система автоматической сборки make

На примере **Задачи №1** рассмотреть назначение, правила создания *make*-файлов.

Для автоматизации сборки приложения есть команда `make` — мощное средство управления проектами. Для работы нужно подготовить `make`-файл, который обычно расположен в каталоге с исходными файлами проекта. Он имеет особый синтаксис и состоит из набора *зависимостей* (а они состоят из целей и набора файлов, от которых зависит цель) и *правил*, которые указывают как создать цель из тех файлов, от которых она зависит. Этот `make`-файл считываетсякомандой `make`, она отслеживает зависимости и сравнивает даты тех файлов, которые должны быть сформированы с теми, от которых они зависят. На основании этого принамаются решения, какие инструкции должны быть выполнены.

Опции команды `make`

Наиболее часто используемые опции:

- `-k` указывает не необходимость продолжать выполнение, если обнаружены ошибки, а не останавливаться при появлении первойпроблемы;
- `-n` указывает на необходимость вывода перечня требуемых действий без реального их выполнения;
- `-f <file>` указывает команде `make` тот файл, который должен восприниматься как `make`-файл. В *Линукс*, если опция не указана, то команда `make` сначала пытается найти и выполнить `GNUmakefile`, затем, есла он не найден, то файл `makefile`, а если и его нет, то файл `Makefile`. Рекомендуется использовать `Makefile`.

Параметра вызова

При простом вызове команды `make` будет выполнено первое задание в команде `make`-файле. Если нужно выполнить какое-то конкретно, то при вызове команды команде `make` нужно указать требуемую цель.

Зависимости

Зависимости определяют, как и с чем связан, от чего зависит целевой файл. Так, для построения исполняемого файла для нашего задания потребуются файлы *`task2.o`* и *`functs.o`*, т. е. оно зависит от них. В свою очередь, файл *`functs.o`* зависит от файлов *`functs.c`* и *`functs.h`* , о он нуждается в повторной компиляции при любом изменении этих файлов. В `make`-файле зависимости указываются так: слева, сначала строки указывается имя задания, двоеточие, пробелы, и затем разделенный пробелами перечень файлов, применяемых для создания выходного файла задания. Пример:

```
main: task2.o functs.o:
task2.o: task2.c functs.h
functs.o: functs.c functs.h
```

Правила

Это второй основной компонент `make`-файла. С помощью заданий указывается, как создать результирующий файл задания из его зависимостей.

Правила указываются после соотетствующей зависимости ивсе правила должны представлять собой строки, начинающиеся со знака **<u>табуляции</u>** (*пробелы не годяться*). Пример:

```
main: task2.o functs.o
     gcc -o main task2.o functs.o -lm

task2.o: task2.c functs.h
     gcc -c task2.c

functs.o: functs.c functs.h
     gcc -c functs.c
```

Комментарии

Комментарии в `make`-файле начинаются со знака # и продолжаются до конца строки.

Макросы

Используются для написания команд в обобщенном виде. Макросы в `make`-файле

задаются в виде конструкции MACRONAME = значение. Затем на значение можно сослаться, указав (MACRONAME) или {MACRONAME}. Можно затать пустое значение длямакроса, оставив пустой часть строки после знака =. Пример:

```
CC = gcc
main: task2.o functs.o
     $(CC) -o main task2.o functs.o -lm
```

Есть еще «встроенный» макрос CFLAGS. Его можно либо определить в make-файле, либо в командной строке при вызове команды make. Утилита во время выполнения правила подставит на место макроса реально заданное при вызове (или указанное в файле) значение. Пример:

```
CC = gcc
main: task2.o functs.o
     $(CC) $(CFLAGS) -o main task2.o functs.o -lm

task2.o: task2.c functs.h
     $(CC) $(CFLAGS) -c task2.c

functs.o: functs.c functs.h
     $(CC) $(CFLAGS) -c functs.c
```

Варианты вызова: `make` `make CFLAGS=-g`

Помощь gcc

Опция -MM для gcc формирует список зависимостей в форме, подходящей для команды make.

Задание 1 без библиотеки: нахождение зависимостей

```
gcc -MM task2.c functs.c
task2.o: task2.c functs.h
functs.o: functs.c functs.h
```

Задание 1 с библиотекой: пример Makefile

```
# Makefile for Task1 with library
CC = gcc
main: task2.o libfun.a
     $(CC) $(CFLAGS) -o main task2.o -L. -lfun -lm

task2.o: task2.c
     $(CC) $(CFLAGS) -c task2.c

libfun.a: functs.o
     ar r libfun.a functs.o
     ranlib libfun.a

functs.o: functs.c functs.h
     $(CC) $(CFLAGS) -c functs.c

clean:
     rm -f *.o *.a main
```

Примеры вызова

```
make
make CFLAGS=-g
make clean
make libfun.a
```

Про это можно говорить долго, на мы рассмотрели самое основное. За остальным — к книгам и документации.

Возможное решение задачи №1

«Однофайловая» версия программы

```
/* Файл task1.c */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

double getX(void);
double f(double x);
double g(double x);

int main(void) {
     double x;

     printf("x: ");
     scanf("%lf", &x);

     printf("f(%f) = %g\t\tg(%f) = %g\n", x, f(x), x, g(x));

     return EXIT_SUCCESS;
}

double f(double x) {
     return exp(-fabs(x))*sin(x);
}

double g(double x) {
     return exp(-fabs(x))*cos(x);
}
```

«Многофайловая» версия программы

```
/* Файл functs.h */
#ifndef _FUNCTS_H_
#define _FUNCTS_H_

double f(double x);
double g(double x);
```

```
#endif
```

/* Файл functs.c */

```
#include <math.h>
#include "functs.h"

double f(double x) {
     return exp(-fabs(x))*sin(x);
}

double g(double x) {
     return exp(-fabs(x))*cos(x);
}
```

/* Файл task2.c */

```
#include <stdio.h>
#include <stdlib.h>
#include "functs.h"


double getX(void);

int main(void) {
     double x;

     x = getX();

     printf("f(%5.2f) = %g\t\tg(%5.2f) = %g\n", x, f(x), x, g(x));

     return EXIT_SUCCESS;
}

double getX(void) {
     double x;
     printf("x: ");
     scanf("%lf", &x);
     return x;
}
```