МИНОБРНАУКИ РОСИИ

Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В. И. Ульянова (Ленина)

ПРОГРАММИРОВАНИЕ

Учебно-методическое пособие

Санкт-Петербург
Издательство СПбГЭТУ «ЛЭТИ»
2018

УДК 004.42(07)
ББК З 973.2–018я7
П78

Авторы: **Кринкин К. В., Берленко Т. А., Заславский М. М., Чайка К. В.**

П78 Программирование: учеб.-метод. пособие. СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2017. 35 с.

ISBN 978-5-7629-2370-5

Содержит материалы по дисциплине «Программирование» и базовые темы по предмету на языке С. Приводятся теоретические материалы, варианты лабораторных работ и вопросы для самоконтроля.

Предназначено для студентов направления «Программная инженерия».

УДК 004.42(07)
ББК З 973.2–018я7

Рецензент – д-р техн. наук, доц. В. В. Езерский (АО «НИИ программных средств).

Утверждено
редакционно-издательским советом университета
в качестве учебно-методического пособия

ISBN 978-5-7629-2370-5

ВВЕДЕНИЕ

В данной работе приводятся методические указания к лабораторным работам по дисциплине «Программирование». Перечень лабораторных работ соответствует рабочей программе дисциплины и содержит следующее:

– компиляция в языке С;

– управляющие конструкции в языке С;

– использование указателей.

– линейные списки;

– обзор стандартной библиотеки;

– динамические структуры данных;

– использование рекурсии.

Информационные технологии (операционные системы, программное обеспечение общего и специализированного назначения, информационные справочные системы) и материально-техническая база, используемые при осуществлении образовательного процесса по дисциплине, соответствуют требованиям федерального государственного образовательного стандарта высшего образования.

Лабораторная работа 1. СБОРКА ПРОЕКТОВ В ЯЗЫКЕ С

1.1. Цель и задачи

Целью данной работы является изучение процесса сборки программ, написанных на языке С на примере использования make-файлов.

Для достижения поставленной цели требуется решить следующие задачи:

1) изучить как происходит процесс компиляции и линковки с использованием компилятора gcc;

2) изучить структуру и правила составления make-файлов;

3) написать make-файл для сборки заданной программы.

1.2. Основные теоретические сведения

Рассмотрим код простой программы, которая выводит на экран сообщение Hello World!:

```
#include <stdio.h>
int main()
{
```

```
printf("Hello World!\n");
return 0;
}
```

Вы можете сохранить код этой программы в файл с любым удобным вам именем и расширением .c, например, main.c. Вы можете скомпилировать ее, выполнив команду:

```
gcc main.c
```

Результатом будет исполняемый файл с именем a.out в текущей директории.

Запустить его можно таким образом:

```
./a.out
```

Чтобы изменить имя исполняемого файла, используйте ключ -o компилятора gcc с указанием нового названия, например:

```
gcc main.c -o main
```

Подробнее о функциях в языке C: (см. 1.1) [1].

1.2.1. Функции

Чтобы лучше понять устройство программы на языке C, рассмотрим базовое понятие «функция». Функция в языке C – подпрограмма (т. е. фрагмент программного кода), которую можно вызвать по имени из другого места программы. Определение функции выглядит следующим образом:

```
<тип_возвращаемого_значения> имя_функции
(список_параметров_функции)
{
<Код_который_исполняется_в_функции>
}
```

Объявление функции сообщает, аргументы каких типов и в каком количестве функция ожидает и какой возвращает результат. Это объявление, называемое прототипом функции, должно быть согласовано с определением и всеми вызовами функции. Если определение функции или вызов не соответствует своему прототипу, возникает ошибка. Обычно объявления функций выносят в специальные заголовочные файлы, имеющие расширение .h. Объявление функции выглядит следующим образом:

```
<тип_возвращаемого_значения> имя_функции
(список_параметров_функции);
```

Если функция не возвращает значения, то <тип_возвращаемого_ значения> указывается void. Если функция не принимает аргументов, то спи-

сок_параметров_функции никак не указывается. Подробнее о функциях в языке C: [1] раздел 1.7.

1.2.2. Главная функция

Выполнение любой программы на языке C начинается с выполнения функции main (говорят также, что main – точка входа в программу). Каждая программа обязательно должна содержать функцию main:

```
#include <stdio.h>
int main()
{
  printf("Hello World!\n");
  return 0;
}
```

В данном случае функция main возвращает целочисленное значение (типа int) и не принимает никаких аргументов (пустые скобки после имени функции).

Для возврата значения из функции служит оператор return. При его вызове выполнение функции немедленно завершается и функция возвращает значение, переданное return. Вы можете переместить эту строку перед строкой с printf и убедиться, что она не будет выполнена.

Значение, которое возвращает функция main, получает та среда, в которой была запущена программа. По соглашению, при корректном завершении программы, функция main должна вернуть значение 0. Ненулевое значение будет расценено операционной системой как код ошибки.

1.2.3. Тело главной функции

Для вывода информации на экран используется библиотечная функция printf. Функции передается строка для печати, заключенная в двойные кавычки. Строка заканчивается символом '\n' - это служебный символ, который отвечает на переход на новую строку. Все служебные символы начинаются со знака обратной черты («бэкслэш» или обратный слэш).

Функция printf содержится в стандартной библиотеке функций, и для ее использования надо явно указать заголовочный файл, в котором она объявлена. Для функции printf, как и для многих других функций ввода/вывода, это заголовочный файл stdio.h(standard input/output):

```
#include <stdio.h>
```

Для стандартных библиотек имя заголовочного файла следует писать в треугольных скобках. Если написать имя заголовочного файла в двойных кавычках например, #include "header_name.h", компилятор будет искать этот файл в директории с исходным кодом.

1.2.4. Препроцессор

Препроцессор – это программа, которая подготавливает код программы для передачи ее компилятору. Команды препроцессора называются директивами и имеют следующий формат:

```
#ключевое_слово параметры
```

Основные действия, выполняемые препроцессором:

– удаление комментариев;

– включение содержимого файлов (#include);

– макроподстановка (#define);

– условная компиляция (#if, #ifdef, #elif, #else, #endif).

1.2.5. #include

Препроцессор обрабатывает содержимое указанного файла и включает содержимое на место директивы. Включаемые таким образом файлы называются заголовочными и обычно содержат объявления функций, глобальных переменных, определения типов данных и другое.

Директива может иметь вид #include "...." либо #include <...>. Для <...> поиск файла осуществляется среди файлов стандартной библиотеки, а для "..." – в текущей директории. Подробнее данная тема изложена в [1] в разделе 4.11.1.

1.2.6 #define

Функция позволяет определить макросы или макроопределения. Имена их принято писать в верхнем регистре через нижние подчеркивания, если это требуется:

```
#define SIZE 10
```

Такое макроопределение приведет к тому, что везде, где в коде будет использовано SIZE, на этапе работы препроцессора это значение будет заменено на 10. Макросы отличаются только наличием параметров:

```
#define MUL_2(x) x*2
```

Таким образом, каждый макрос MUL_2 в коде будет преобразован в выражение x*2, где x – его аргумент.

Следует обратить особое внимание, что define выполняет просто подстановку идентификатора (без каких-то дополнительных преобразований), что иногда может приводить к ошибкам, которые трудно найти. Подробнее о работе defne можно узнать в в 4.11.2 [1].

1.2.7. #if, #ifdef, #elif, #else, #endif

Директивы условной компиляции допускают возможность выборочной компиляции кода. Это может быть использовано для настройки кода под определенную платформу, внедрения отладочного кода или проверки на повторное включение файла. Данные директивы описаны в 4.11.3 [1].

Оператор ## используется для объединения двух лексем, что может быть полезным.

1.2.8. Компиляция

Компиляция – процесс преобразования программы с исходного языка высокого уровня в эквивалентную программу на языке более низкого уровня (в частности, в машинном языке).

Компилятор – программа, которая осуществляет компиляцию (рис. 1.1).

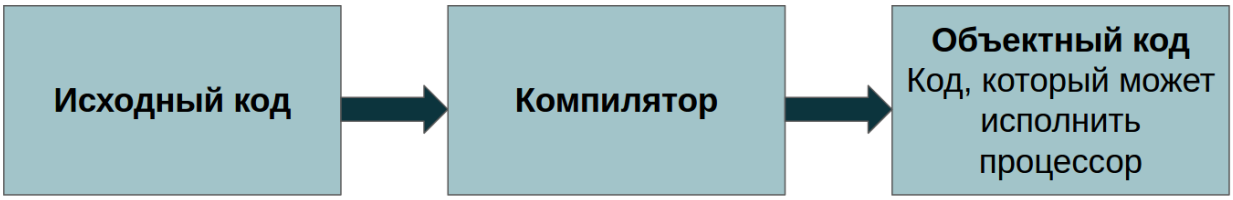

Рис. 1.1. Принцип работы компилятора

Большая часть компиляторов преобразует программу в машинный код, который может быть выполнен непосредственно процессором. Этот код различается между операционными системами и архитектурами. Однако, в некоторых языках программирования программы преобразуются не в машинный, а в код на более низкоуровневом языке, но подлежащий дальнейшей интерпретации (байт-код). Это позволяет избавиться от архитектурной зависимости, но влечет за собой некоторые потери в производительности.

Компилятор языка C принимает *исходный текст* программы, а результатом является *объектный модуль*. Он содержит в себе подготовленный код,

который может быть объединен с другими объектными модулями при помощи линковщика для получения готового *исполняемого модуля*.

1.2.9. Сборка программ

Сборка программы из исходных кодов в исполняемый файл включает в себя два основных этапа: компиляция и линковка. Сначала для каждого файла исходного кода компилятор генерирует объектный файл, а после линковщик преобразует все объектные файлы в исполняемый. Наглядно данные этапы показаны на рис. 1.2.

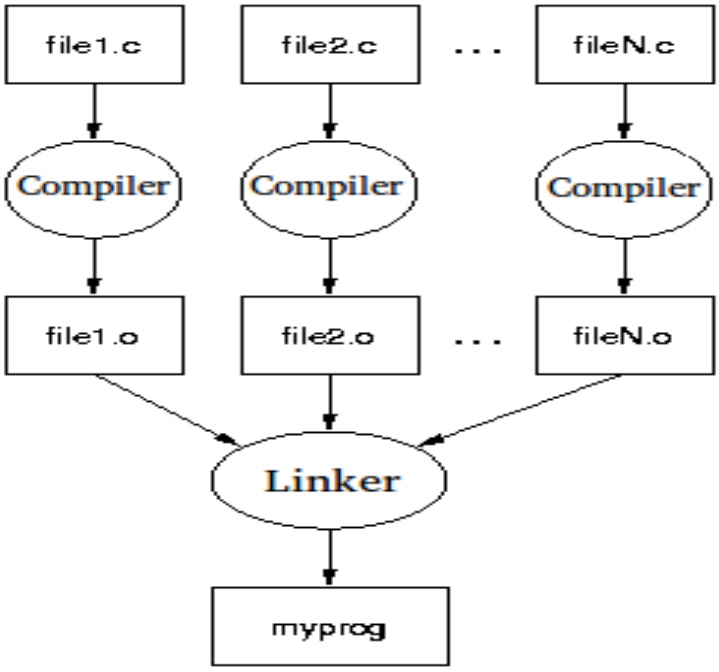

Рис. 1.2. Процедура сборки программы из исходных кодов

Для получения объектного файла следует вызвать gcc с ключом –c:

```
gcc -c test.c
```

Это приведет к созданию объектного файла с расширением .o, но с тем же именем, что у файла с исходным кодом.

Для линковки достаточно вызвать команду gcc с указанием объектных файлов:

```
gcc test.o test1.o -o testprog
```

1.2.10. Make-файлы

Для сборки программ, состоящих из нескольких файлов, часто используются make-файлы. Make-файл – это текстовый файл, в котором определенным образом описаны правила сборки приложения. Для работы с make-файлами используется утилита make, которая позволяет не компилировать

файлы дважды, т. е. в тех случаях, когда в вашем большом проекте изменился, например, только один файл, утилита скомпилирует только его и не станет перекомпилировать все остальные. Давайте рассмотрим основные принципы работы make.

При запуске утилита пытается найти файл с заданным по умолчанию именем Makefile в текущем каталоге и выполнить содержащиеся в нем инструкции. Возможно явно указать, какой make-файл использовать с помощью ключа -f:

```
make -f MyMakefile
```

Базовые части make-файла выглядят обычно следующим образом:
– цель: зависимости;
– [tab] команда (таких строк может быть несколько).

По умолчанию основной целью считается первая цель.

Целью в make-файле является файл, который получается в результате выполнения команд. Также целью может быть название действия, которое будет выполнено (без зависимостей), например:

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

Зависимости – это файлы, которые make проверяет на наличие и дату изменений. Зависимости необходимы для получения цели: утилита make проверяет, были ли зависимости обновлены с последнего запуска make, и если зависимость стала новее, обновляет цель. Таким образом обновляются только «устаревшие» цели, и нет необходимости каждый раз пересобирать весь проект.

Пример make файла для сборки программы, которая состоит из двух файлов: main.c и hello.c (заголовочные файлы отсутствуют):

```
hello: main.o  hello.o
        gcc main.o hello.o -o hello
main.o: main.c
        gcc -c main.c
hello.o: hello.c
        gcc -c hello.c
```

Данный файл следует читать следующим образом: основной целью сборки является цель hello, которая зависит от целей main.o hello.o. Сначала

будут выполнены зависимые цели, а после – команды для выполнения основной.

1.2.11. Программа для считывания строки с консоли и вывода строки на консоль

Заголовочный файл стандартной библиотеки, который содержит функции консольного ввода/вывода:

stdio.h

Заголовочный файл стандартной библиотеки, который содержит функции обработки строк и управления памятью:

string.h

Прототип функции вывода строки str:

```
int puts(const char *str);
```

Прототип функции конкатенации строк:

```
char * strncat( char * destptr, char * srcptr, size_t num );
```

Функция добавляет первые num символов строки srcptr к концу строки destptr, плюс символ конца строки. Если строка srcptr больше, чем количество копируемых символов num, то после скопированных символов неявно добавляется символ конца строки.

Описание функции для ввода массива символов name (предполагается, что строка не содержит более 80 символов):

```
char* get_name(){
    char* name = (char*)malloc(80*sizeof(char));
    int i = 0;
    char ch;
    while ((ch = getchar()) != '\n')
    {
      name[i] = ch;
      i++;
    }
    name[i] = '\0';
    return name;
}
```

Описание главной функции:

```
int main(){
    char hello[90] = "Hello, ";
    char* result;
    result = get_name();
    print_str(strncat(hello, result, 80));
    free(result);
    return 0;
}
```

1.3. Общая формулировка задачи

Требуется создать проект, который состоит из пяти файлов: main.c, print_str.c, get_name.c, print_str.h, get_name.h в каталоге, имя которого содержит ваше имя, фамилию и номер лабораторной (например, ivanov_ivan_1).

Файл get_name.c должен содержать описание функции, которая считывает из входного потока имя пользователя и возвращает его.

Файл get_name.h должен содержать прототип функции, которая считывает из входного потока имя пользователя и возвращает его.

Файл print_str.c должен содержать описание функции, которая принимает в качестве аргумента строку и выводит ее (функция ничего не возвращает).

Файл print_str.h должен содержать прототип функции, которая принимает в качестве аргумента строку и выводит ее (функция ничего не возвращает).

Файл main.c содержит главную функцию, которая вызывает функцию из файла get_name.h, добавляет к результату выполнения функции строку Hello, и передает полученную строку в функцию вывода строки из print_str.h.

После того как ваш проект будет готов, создайте для него Makefile.

1.4. Описание последовательности выполнения работы

В данной работе необходимо правильно разбить проект по файлам, подключить заголовочные файлы, используя директивы препроцессора, и написать отчет по проделанной работе.

1.5. Пример выполнения задания

Задание: написать make-файл для программы, состоящей из трех файлов: print.c, print.h и main.c.

Содержимое файлов:

print.c

```
void print(int a, int b){
    int c = a + b;
    print(a, b);
    return 0;
}
```

print.h

```
void print(int a, int b);
```

main.c

```
#include "print.h"
int main(){
    int a = 10;
    int b = 20;
    print(a, b);
    return 0;
}
```

makefile

```
example: main.o  print.o
  gcc main.o print.o -o example

main.o: main.c
        gcc -c main.c

print.o: print.c
  gcc -c print.c
```

1.6. Вопросы для контроля

1. Как выполняется процесс сборки программы на языке C?
2. Что такое «зависимости» в make-файле?
3. Что такое компиляция?
4. Зачем нужен линковщик?

Лабораторная работа 2. УПРАВЛЯЮЩИЕ КОНСТРУКЦИИ ЯЗЫКА С

2.1. Цель и задачи

Целью работы является освоение работы с управляющими конструкциями на языке С на примере использующей их программы.

Для достижения поставленной цели требуется решить следующие задачи:

– ознакомиться с существующими управляющими конструкциями;

– научиться их использовать;

– написать программу, решающую задачу в соответствии с индивидуальным условием.

2.2. Основные теоретические сведения

2.2.1. Типы данных

В языке С имеется несколько основных типов:

– char – один байт (обычно используется для хранения символов);

– int – целое число;

– float – вещественное число;

– double – вещественное число двойной точности.

Также есть ряд квалификаторов:

– short – короткое;

– long – длинное;

– unsigned – беззнаковое.

Влияние спецификаторов размера типа (short, long) зависит от конкретной архитектуры.

Специального типа данных для хранения символьных строк в языке С нет. Для хранения строк используются массивы типа char, в которых после последнего символа строки находится нулевой символ '\0'.

Массив – структура данных, в виде набора элементов одного типа, расположенные в памяти друг за другом.

В языке С присутствуют привычные арифметические операции: +, -, *, / и операция %.Дело в том, что операция деления (/) над целыми числами выполняется с отбрасыванием целой части. Операция % позволяет получить остаток от деления (и допустима только над целыми).

Операции отношения: >=, >, ==, !=, <=, <.

Логические операции:

! – логическое НЕ,

&& - логическое И,

|| – логическое ИЛИ.

Логические выражения вычисляются «ленивым» образом слева направо. Таким образом, как только становится ясным результат выражения, вычисление его аргументов прекращается.

В языке С предусмотрены специальные операции увеличения (++) и уменьшения (– –). Есть префиксная и постфиксная их версии: отличие их в том, что префиксная операция выполняется до использования переменной в выражении, а постфиксная – после. Про приоритет операций рекомендуем почитать подробнее в 2.12 [1].

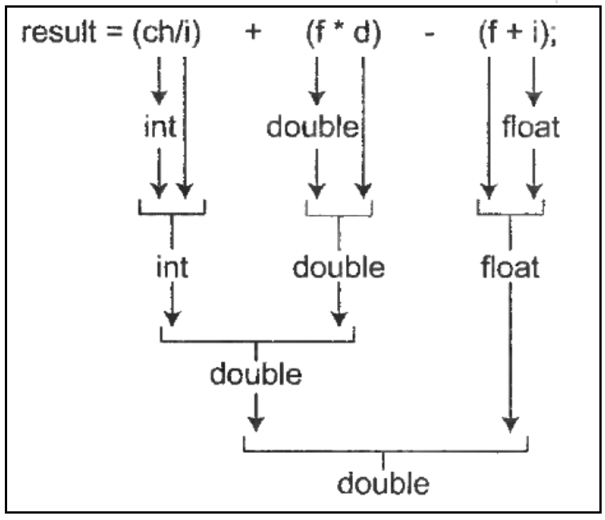

Рис. 2.1. Процедура преобразования типов в выражении

Если в выражениях встречаются операнды различных типов, то они преобразуются к общему типу в соответствии с небольшим набором правил. Автоматически производятся только преобразования, имеющие смысл, как, например, преобразование целого в плавающее в выражениях типа f+i. Выражения же, лишенные смысла, такие как использование переменной типа float в качестве индекса, запрещены.

Пример преобразования типов в выражении (подробнее рекомендуем почитать в 2.7 [1]) изображен на рис. 2.1, где:

– char – ch;

– int – i;

– float – f;

– double – d.

2.2.2. Поразрядные (побитовые) операции

В языке С существует возможность оперировать отдельными битами (применимо только к целым типам данных).Для этого есть следующие логические операции:

& – побитовое И,

| – побитовое ИЛИ,

^ – побитовое исключающее ИЛИ,
<< – побитовый сдвиг влево,
>> – побитовый сдвиг вправо
~ – дополнение (унарная операция).

Подробнее об этих операциях можно узнать в 2.9 [1].

2.2.3 Основные операторы в языке C

Операторный блок:

{[<*оператор 1*>...<*оператор N*>]}

Действие заключается в группировке операторов в единый блок.

Условный оператор:

if (<*выражение*>) <*оператор 1*> [**else** <*оператор 2*>]

Если выражение интерпретируется как истина, то оператор1 выполняется; может иметь необязательную ветку else, путь выполнения программы пойдет в случае, если выражение ложно.

Оператор множественного выбора

```
switch (<выражение>)
  { case <константное выражение 1>: <операторы 1>
    case <константное выражение 2>: <операторы 2>
    ...
    case <константное выражение N>: <операторы N>
    [default: <операторы>]
    }
```

выполняет поочередное сравнение выражения со списком константных выражений. При совпадении выполнение программы начинается с соответствующего оператора. Если совпадений не было, выполняется необязательная ветка default. Важно помнить, что операторы после первого совпадения будут выполняться далее один за другим. Чтобы этого избежать, следует использовать оператор break.

Цикл с предусловием:

while (<выражение>) <оператор>

На каждой итерации цикла происходит вычисление выражения и если оно истинно, то выполняется тело цикла.

Цикл с постусловием:

do <оператор> **while** <выражение>;

На каждой итерации цикла сначала выполняется тело цикла, а после вычисляется выражение. Если оно истинно, выполняется следующая итерация.

Цикл со счетчиком:

```
for ([<начальное выражение>];
     [<условное  выражение>];
     [<выражение приращения>])
   <оператор>
```

Условием продолжения цикла (как и в цикле с предусловием) является некоторое выражение, однако в цикле со счетчиком есть еще 2 блока – начальное выражение, выполняемое один раз перед первым началом цикла и выражение приращения, выполняемое после каждой итерации цикла.

Оператор break – досрочно прерывает выполнение цикла.

Оператор continue – досрочный переход к следующей итерации цикла.

2.2.4. PRINTF

Управляющая строка содержит два типа объектов: обычные символы, которые просто копируются в выходной поток, и спецификации преобразований, каждая из которых вызывает преобразование и печать очередного аргумента printf. Каждая спецификация преобразования начинается с символа % и заканчивается символом преобразования. Выражение

```
int printf ( const char * format, arg1, arg2, ...argN)
```

преобразует, определяет формат и печатает свои аргументы в стандартный поток вывода под управлением строки *format.*

Функция возвращает количество успешно выведенных символов. Подробнее о символах преобразования и символах, которые могут находиться перед ними, следует ознакомиться в 7.3 [1].

2.2.5. SCANF

Управляющий аргумент описывается ниже; другие аргументы, каждый из которых должен быть указателем, определяют, куда следует поместить соответствующим образом преобразованный ввод. Выражение

```
int scanf ( const char * format, arg1, arg2, ...argN)
```

читает символы из стандартного ввода, интерпретирует их в соответствии с форматом, указанном в аргументе control, и помещает результаты в остальные аргументы.

Управляющая строка обычно содержит спецификации преобразования, которые используются для непосредственной интерпретации входных последовательностей.

Функция возвращает количество успешно считанных аргументов. Подробнее о символах управляющей строки можно прочесть в разделе 7.4 [1].

2.2.6. Отладка с помощью логов

Зачастую ваши программы могут успешно компилироваться, но при этом работают не так как вы ожидали, либо аварийно завершаются. Чтобы устранить проблему, необходимо прежде всего ответить на вопрос: в какой строке исходного кода происходит неправильное поведение? Наиболее простым способом является логгирование – добавление в программу специальных вызовов printf, которые протоколируют ход ее работы:

printf("%s, %s, %d: ваше сообщение\n", __FILE__, __func__, __LINE__),

где __FILE__ , __func__ , __LINE__ – специальные макросы, которые заменяются препроцессором на имя полного пути к файлу, название функции и номер строки.

Пример:

```
#include <stdio.h>

int main()
{
   int N=3, result;
   printf("%s, %s, %d: Инициализация переменных\n", __FILE__,
__func__, __LINE__);
   result = N;
   printf("%s, %s, %d: Присваивание result\n", __FILE__, __func__,
__LINE__);
   result = result*N;
   printf("%s, %s, %d: Умножение на N\n", __FILE__, __func__,
__LINE__);
   return 0;
}
```

Таким образом, если при работе программы будет сбой, то можно будет по ее выводу определить, на какой именно строке произошел сбой. Подробнее о данных макросах можно прочитать в [2] и [3].

2.2.7. Цветной вывод в командной строке

При активном использовании логов неизбежно возникает следующая проблема – при достаточно интенсивном выводе на консоль становится сложно выделить важную информацию. Для решения этой проблемы существует стандартный механизм – вывод цветных строк в командную строку. Рассмотрим пример, который выводит фразу Hello, world! в двух цветах (красном и голубом):

```
#include <stdio.h>

#define RED   "\033[0;31m"
#define CYAN  "\033[0;36m"
#define NONE   "\033[0m"

int main()
{
printf("%sHello, %sworld!%s\n", RED, CYAN, NONE);
return 0;
}
```

В примере в консоль на самом деле выводится следующая строка:

```
\033[0;31mHello, \033[0;36mworld!\033[0m.
```

Указание на то, какой цвет использовать, делается с помощью добавления специального кода в выводимую на консоль строку. Код цвета представляет собой команду для терминала: «переключи текущий цвет на цвет N». Существует также специальный код для возврата к цвету по умолчанию: "\033[0m" . Данную команду необходимо использовать после любых манипуляций с цветом, для того чтобы не повлиять на корректность отображения вывода других приложений в данной сессии терминала.

Кодирование цветов осуществляется следующим образом:

```
\033[STYLE;BACKGROUND_COLOR;FOREGROUND_COLORm,
```

где STYLE это стиль отображения (0 – обычный, 1 – полужирный, 4 – подчеркнутый, 9 – зачеркнутый), BACKGROUND_COLOR и FOREGROUND_COLOR – коды цвета из таблицы в [4] для фона и текста соответственно. При

этом можно по желанию пропускать один или несколько этих атрибутов, например:

```
\033[STYLE;BACKGROUND_COLORm
\033[STYLE;FOREGROUND_COLORm
```

Пример, в котором весь текст выводится красным цветом, но при этом фон у первого слова – по умолчанию, а у второго – голубой:

```
#include <stdio.h>
#define FIRST   "\033[0;31m"
#define SECOND  "\033[46;31m"
#define NONE    "\033[0m"

int main()
#include <stdio.h>
#define FIRST   "\033[0;31m"
#define SECOND  "\033[46;31m"
#define NONE    "\033[0m"

int main()
{
printf("%sHello, %sworld!%s\n", FIRST, SECOND, NONE);
return 0;
}{
printf("%sHello, %sworld!%s\n", FIRST, SECOND, NONE);
return 0;
}
```

С дополнительными материалами можно ознакомиться в [4], [5].

2.3. Общая формулировка задачи

В текущей директории создайте проект с make-файлом. Главная цель должна приводить к сборке проекта. Файл, который реализует главную функцию, должен называться menu.c; исполняемый файл – menu. Определение каждой функции должно быть расположено в отдельном файле, название файлов указано в скобках около описания каждой функции.

Реализуйте функцию-меню, на вход которой подается одно из значений 0, 1, 2, 3 и массив целых чисел размера не больше 20. Числа разделены пробелами. Строка заканчивается символом перевода строки.

2.4. Перечень индивидуальных заданий

1. В зависимости от значения, функция menu должна выводить следующее:

0 – индекс первого отрицательного элемента. (index_first_negative.c);

1 – индекс последнего отрицательного элемента. (index_last_negative.c);

2 – Найти произведение элементов массива, расположенных от первого отрицательного элемента (включая элемент) и до последнего отрицательного (не включая элемент). (multi_between_negative.c);

3 – Найти произведение элементов массива, расположенных до первого отрицательного элемента (не включая элемент) и после последнего отрицательного (включая элемент). (multi_before_and_after_negative.c).

Иначе необходимо вывести строку «Данные некорректны».

2. В зависимости от значения, функция menu должна выводить следующее:

0 – максимальное число в массиве. (max.c);

1 – минимальное число в массиве. (min.c);

2 – разницу между максимальным и минимальным элементом. (diff.c);

3 – сумму элементов массива, расположенных до минимального элемента. (sum.c).

Иначе необходимо вывести строку «Данные некорректны».

3. В зависимости от значения, функция menu должна выводить следующее:

0 – индекс первого нулевого элемента. (index_first_zero.c);

1 – индекс последнего нулевого элемента. (index_last_zero.c);

2 – Найти сумму модулей элементов массива, расположенных от первого нулевого элемента и до последнего. (sum_between.c);

– 3 : Найти сумму модулей элементов массива, расположенных до первого нулевого элемента и после последнего. (sum_before_and_after.c).

Иначе необходимо вывести строку «Данные некорректны».

2.5. Описание последовательности выполнения работы

В данной работе необходимо написать программу на языке С и составить отчет.

2.6. Пример выполнения задания

Задание: напишите функцию is_in, которая определит попадание точки в закрашенную область. На вход функция получает два числа x и y. Эти числа имеют точность 10^{-1} . Функция должна вывести строку "true", если точка попадает в заштрихованную область или попадает на границу, и строку "false" в противном случае (рис 2.2).

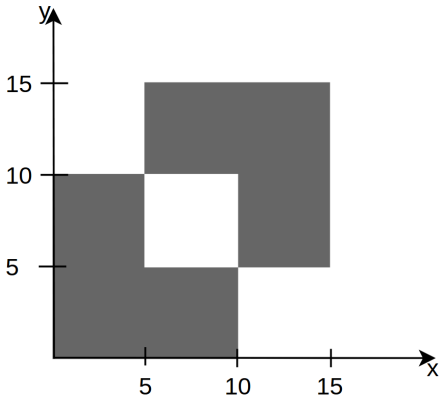

Рис. 2.2. Пример области

Код на языке С:

```
void is_in(double x, double y){
 int in_1st_square = 0;
 int in_2nd_square = 0;
 int in_3rd_square = 0;

 if( x>=0 && x<=10 &&
    y>=0 && y<=10 ){
  in_1st_square = 1;
 }

 if( x>=5 && x<=15 &&
    y>=5 && y<=15 ){
  in_2nd_square = 1;
 }

 if( x>5 && x<10 &&
    y>5 && y<10 ){
  in_3rd_square = 1;
 }

 if(     (in_1st_square     ||     in_2nd_square)     &&
```

```
!in_3rd_square )
                printf("true");
              else
                printf("false");
            }
```

2.7. Вопросы для контроля

1. Тело какого цикла выполнится всегда как минимум один раз?

2. Каково будет поведение программы, если опустить оператор break в ветках оператора switch?

Лабораторная работа 3. ИСПОЛЬЗОВАНИЕ УКАЗАТЕЛЕЙ

3.1. Цель и задачи

Целью работы является освоение работы с указателями и динамической памятью.

Для достижения поставленной цели требуется решить следующие задачи:

– ознакомиться с понятием «указатель»;

– научиться использовать указатели в языке С;

– изучить способы работы с динамической памятью в языке С;

– написать программу с использованием динамической памяти в соответствии с индивидуальным заданием.

3.2. Основные теоретические сведения

Указатель – некоторая переменная, значением которой является адрес в памяти некоторого объекта, определяемого типом указателя. Для работы с указателями используется 2 оператора:

* – оператор разыменования;

& – оператор взятия адреса.

Объявляется указатель (аналогичным образом как и переменная), но перед именем указывается *:

```
int *p;
int a = 10;
p = &a;
*p = 25;
```

Таким образом, после выполнения данного фрагмента кода, значение переменной «a» будет равно 25, потому что *p следует трактовать как «обратиться к ячейке по адресу, который хранится в p».

Давайте рассмотрим, как связаны указатели и массивы в языке C. Любую операцию, которую можно выполнить с помощью индексов массива, можно сделать и с помощью указателей.

Пусть у нас объявлен массив из 10 элементов типа int:

```
int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

и указатель на int:

```
int* p_array.
```

Тогда запись вида:

```
p_array = &array[0]
```

означает, что p_array указывает на нулевой элемент массива A, т. е. p_array содержит адрес элемента array[0].

Если p_array указывает на некоторый определенный (array[i]) элемент массива array, то p_array+1 указывает на следующий элемент после p_array, т. е. на array[i+1].

Если указатели p и q указывают на элементы одного массива, то к ним можно применять операторы отношения ==, !=, <, >= и т. д. Например, отношение вида p < q истинно, если p указывает на более ранний элемент массива, чем q. Любой указатель всегда можно сравнить на равенство и неравенство с нулем. А вот для указателей, не указывающих на элементы одного массива, результат арифметических операций или сравнений не определен.

3.2.1. Передача аргумента в функцию

До сих пор мы писали свои функции, в которые передавали аргументы по значению. Это значит, что если мы передали аргумент в функцию и изменили его там, в вызывающей функции изменения не сохранятся. Это происходит, поскольку функция получает копию аргумента и не может повлиять на его оригинал.

Рассмотрим функцию swap, которая должна менять значения целых переменных:

```
void swap(int a, int b){ // НЕПРАВИЛЬНЫЙ ВАРИАНТ
int c = a;
a = b;
b = c;
}
```

Как мы уже знаем, в данном случае функция swap получает копии аргументов a и b, поэтому, если мы вызовем ее где-нибудь в другой функции, например, main, значения аргументов a и b не изменятся:

```
int main(){
    int a = 10;
    int b = 100;
    swap(a, b);
    printf("%d %d", a, b); // 10 100
    return 0;
}
```

Для того, чтобы функция swap могла модифицировать свои аргументы, несколько ее изменим:

```
void swap(int* a, int* b){
    int c = *a;
    *a = *b;
    *b = c;
}
```

После изменения она принимает два указателя на переменные типа int и модифицирует значения, на которые они ссылаются.

Теперь при вызове функции, теперь ей надо передать адреса аргументов:

```
swap(&a, &b);
```

Динамическое выделение памяти может быть удобно для создания массивов, размер которых на момент написания программы неизвестен и определяется только в процессе выполнения программы.

Для этого используется функция malloc (для ее использования следует подключить заголовочный файл stdlib.h):

```
void * malloc( size_t sizemem ).
```

Функция выделяет из памяти sizemem и возвращает указатель на выделенную память, который следует привести к требуемому типу.

Для изменения размера выделенной ранее динамически области памяти используется функция realloc:

```
void * realloc( void * ptrmem, size_t size )
```

Она получает указатель на выделенную ранее область памяти и новый ее размер (он может быть как увеличен, так и уменьшен) и возвращает указатель на область памяти измененного размера (при изменении размера области памяти ее начальный адрес может измениться). Следовательно, функция realloc может выполняться в некоторых случаях довольно долго, поэтому не следует использовать ее слишком часто без явной необходимости.

Если выделенная память больше не требуется, следует обязательно высвобождать ее с помощью оператора free.

Пример:

```
int *p = (int *) malloc(5 * sizeof(int));
p = (int *) realloc(p, 10 * sizeof(int));
free(p);
```

Формально в языке С нет специального типа данных для строк, но представление их довольно естественно. Строки в языке С – это массивы символов, завершающиеся нулевым символом ('\0'). Это порождает следующие особенности, которые следует помнить:

– нулевой символ является обязательным;

– символы, расположенные в массиве после первого нулевого символа никак не интерпретируются и считаются мусором;

– отсутствие нулевого символа может привести к выходу за границу массива;

– фактический размер массива должен быть на единицу больше количества символов в строке (для хранения нулевого символа);

– выполняя операции над строками, нужно учитывать размер массива, выделенный под хранение строки;

– строки могут быть инициализированы при объявлении.

Примеры:

```
char s0[] = "Hello!";   // массив длины 7
char s1[50] = "World";   // массив длины 50, из которых используется 6 ячеек
```

```
s1[3] = '\0';                 // вместо 'l' теперь '\0'
printf("%s\n",s0);          // выведет "Hello!"
printf("%s\n",s1);          // выведет "Wor"
```

Для считывания строки рекомендуется использовать функцию fgets:

char* fgets(char *str, int num, FILE *stream)

Она принимает в качестве аргументов массив, в который следует записать строку, размер массива и источник, откуда следует считывать (для считывания из консоли следует указать stdin).

В отличие от функций scanf и gets, функция fgets гарантирует, что не будет считано больше num-1 символов (что не приведет к выходу за границы массива) и строка будет завершена корректно (нулевым символом).

Как вы уже могли догадаться, если строка в С – массив символов, то массив строк – это двумерный массив символов, где каждая строка – массив, хранящий очередную символьную строку.

3.3. Общая формулировка задачи

Напишите программу, которая форматирует некоторый текст и выводит результат на консоль. На вход программе подается текст неизвестной длины, который заканчивается предложением «Dragon flew away!». Предложение (кроме последнего) может заканчиваться следующим образом.

. (точка);

; (точка с запятой);

? (вопросительный знак).

Программа должна изменить и вывести текст следующим образом:

– все предложения, которые заканчиваются на '?', должны быть удалены;

– каждое предложение должно начинаться с новой строки;

– табуляция в начале предложения должна быть удалена.

Текст должен заканчиваться фразой: «Количество предложений до n и количество предложений после m», где n – количество предложений в изначальном тексте (без учета терминального предложения «Dragon flew away!») и m – количество предложений в отформатированном тексте (без учета предложения про количество из данного пункта).

Порядок предложений не должен меняться. Статически выделять память под текст нельзя.

3.4. Перечень индивидуальных заданий

1. Все предложения, которые заканчиваются на '?' должны быть удалены.

2. Все предложения, в которых есть цифра 7, должны быть удалены.

3. Все предложения, в которых есть цифры внутри слов, должны быть удалены (это не касается предложений, которые начинаются/заканчиваются цифрами).

4. Все предложения, в которых есть число 555, должны быть удалены.

5. Все предложения, в которых больше одной заглавной буквы, должны быть удалены.

3.5. Описание последовательности и пример выполнения работы

В данной работе необходимо написать программу на языке С и составить отчет.

Задание: напишите программу, которая принимает на вход строку, удаляет из нее все гласные латинские буквы как в верхнем, так и нижнем регистре, печатает результат на экран.

Программа на языке С:

```
#include <stdio.h>
#include <ctype.h> // заголовочный файл библиотеки для работы с символами
char vowels[]={'A', 'E', 'I', 'O', 'U', 'Y'}; // массив с гласными буквами
// фунция проверки символа на то, гласный ли он
int isVowel(char ch)
{
  int i;
  for(i=0; i<sizeof(vowels); i++) // проверяем с каждой гласной из массива
    if(toupper(ch) == vowels[i]) // переводим символ в верхний регистр и сравниваем
       return 1;
  return 0; // если совпадений не было - это согласная
}

int main() {

  char s_in[100];
  char s_out[100];
```

```
    int i; // индекс текущей ячейки для исходной строки
    int j; // индекс текущей ячейки для строки-результата
    fgets(s_in, 100, stdin); // аргументы: буфер, размер буфера, стандартный поток ввода
    for(i=0, j=0; s_in[i]; i++) // пока s_in[i] не нулевой символ
     if(!isVowel(s_in[i])) // если согласная
       s_out[j++] = s_in[i];    // записываем его в очередную ячейку результата, переходим к следующей
    s_out[j] = '\0'; // не забываем завершить строку-результат нулевым символом
    printf("%s", s_out);

    return 0;
}
```

3.7. Вопросы для контроля

1. Что хранится в переменной типа указатель на int?
2. Почему частое использование функции realloc может замедлять выполнение программы?

Лабораторная работа 4. ЛИНЕЙНЫЕ СПИСКИ

4.1. Цель и задачи

Целью работы является освоение работы с линейными списками.

Для достижения поставленной цели требуется решить следующие задачи:

– ознакомиться со структурой «список»;

– ознакомиться со списком операций используемых для списков;

– изучить способы реализации этих операций на языке С;

– написать программу реализующую двусвязный линейный список и решающую задачу в соответствии с индивидуальным заданием.

4.2. Основные теоретические сведения

Линейный односвязный список – это структура данных, представляющая собой список узлов, каждый из которых хранит какие-то полезные данные и указатель на следующий (если список двусвязный, то и на предыдущий) элемент (рис. 4.1).

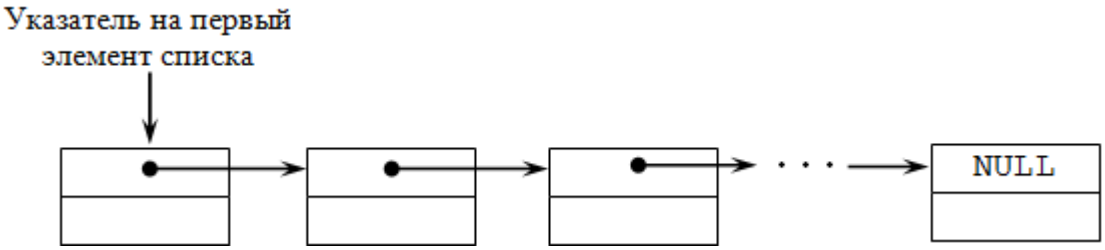

Рис. 4.1. Линейный односвязный список

Список можно рассматривать как более гибкую альтернативу массиву: порядок элементов в списке не связан с их расположением в памяти. Это порождает, например, то, что такие операции как поиск и удаление не связаны со сдвигами остальных элементов, однако все операции со списком являются исключительно последовательными – прямой доступ к элементам списка невозможен.

Рассмотрим пример создания линейного односвязного списка, состоящего из двух элементов.

Пусть нам дана структура Node следующего вида:

```
struct Node{
    int x;
    int y;
    float r;

   struct Node* next; // указатель на следующий элемент
};
```

Проинициализируем два элемента списка в функции main():

```
int main(){
    struct Node * p1 = (struct Node*)malloc(sizeof(struct Node));
    struct Node * p2 = (struct Node*)malloc(sizeof(struct Node));

    p1->x = 2; // используем оператор -> поскольку p1 - это указатель на
структуру Node
    p1->y = 2;
    p1->r = 2.5;

    p2->x = 5;
    p2->y = 5;
    p2->r = 5.5;
```

```
    p1->next = p2; // связываем указатель на следующий элемент у p1 и
p2
    p2->next = NULL; // указатель на следующий элемент для p2 NULL

    free(p1); // освобождение памяти
    free(p2);

    return 0;
}
```

У нас получился линейный список из двух элементов: p1 и p2.

Для работы со списками используются основные функции:

– вставка элемента в конец списка/голову списка;

– вставка после определенного элемента;

– определение количества элементов списка;

– удаление элемента из конца списка/из головы;

– удаление определенного элемента списка.

Рекомендуется реализовывать все нужные функции работы со структурами отдельно в каждой функции, это делает код более читаемым и позволяет избежать дублирования.

4.3. Общая формулировка задачи

Создайте двунаправленный список музыкальных композиций MusicalComposition и api (application programming interface – в данном случае набор функций) для работы со списком.

Структура элемента списка (тип – MusicalComposition):

– name; строка неизвестной длины (гарантируется, что длина не может быть больше 80 символов), название композиции;

– author; строка неизвестной длины (гарантируется, что длина не может быть больше 80 символов), автор композиции/музыкальная группа;

– year; целое число, год создания.

Функция для создания элемента списка (тип элемента MusicalComposition).

MusicalComposition* createMusicalComposition(char* name, char* author, int year).

Функции для работы со списком:

MusicalComposition* createMusicalCompositionList(char** array_names, char** array_authors, int* array_years, int n); // создает список музыкальных композиций MusicalCompositionList, в котором:

– n – длина массивов array_names, array_authors, array_years;

– поле name первого элемента списка соответствует первому элементу списка array_names (array_names[0]);

– поле author первого элемента списка соответствует первому элементу списка array_authors (array_authors[0]);

– поле year первого элемента списка соответствует первому элементу списка array_authors (array_years[0]).

Аналогично для второго, третьего, ... n−1-го элемента массива.

Длина массивов array_names, array_authors, array_years одинаковая и равна n, это проверять не требуется.

Функция возвращает указатель на первый элемент списка.

void push(MusicalComposition* head, MusicalComposition* element); // добавляет element в конец списка musical_composition_list.

void removeEl (MusicalComposition* head, char* name_for_remove); // удаляет элемент element списка, у которого значение name равно значению name_for_remove.

int count(MusicalComposition* head); //возвращает количество элементов списка.

void print_names(MusicalComposition* head); //выводит названия композиций.

4.4. Описание последовательности и пример выполнения работы

В данной работе необходимо написать программу на языке С и составить отчет.

Задание: напишите функцию, которая создает линейный односвязный список из 10 элементов. Элемент списка – структура вида

```
struct Node{
    int x;
    int y;
    float r;
    struct Node* next;
};
```

Программа на языке С:

```
typedef struct Node{
   int x;
   int y;
   float r;
   struct Node* next;
}Node;

int main(){
  Node* head = (Node*)malloc(sizeof(Node));
  Node* tmp = (Node*)malloc(sizeof(Node));
  head->next = tmp;
  for(i = 1; i < 10; i++){
     tmp->next = (Node*)malloc(sizeof(Node));
     tmp->next->next = NULL;
     tmp = tmp->next;
  }
}
```

4.5. Вопросы для контроля

1. В чем отличие линейного списка от массива?
2. Какие обязательные поля должны быть в каждом узле двусвязного линейного списка?
3. Чем отличается двусвязный список от односвязного?

СПИСОК ЛИТЕРАТУРЫ

1. Керниган Б., Ритчи Д. Язык программирования С. 3-е изд. СПб.: Невский Диалект. 2004.
2. The C Preprocessor : Standard Predefined Macros .
URL: https://gcc.gnu.org/onlinedocs/cpp/Standard-Predefined-Macros.html
3. Logging with GCC (материал повышенной сложности).
URL: http://www.valvers.com/programming/c/logging-with-gcc/
4. ANSI escape code- From Wikipedia, the free encyclopedia.
URL: https://en.wikipedia.org/wiki/ANSI_escape_code#Colors
5. C Programming / Linux – Color text output.
URL: https://ramprasadk.wordpress.com/2010/06/09/c-programming-linux-color-text-output/

СОДЕРЖАНИЕ

Кирилл Владимирович Кринкин
Татьяна Андреевна Берленко
Марк Маркович Заславский
Константин Владимирович Чайка

Программирование

Учебно-методическое пособие

Редактор М. Б. Шишкова

Подписано в печать 03.12.18. Формат 60×84 1/16.
Бумага офсетная. Печать цифровая. Печ. л. 2,25
Гарнитура «Times New Roman». Тираж 77 экз. Заказ .

Издательство СПбГЭТУ «ЛЭТИ»
197376, С.-Петербург, ул. Проф. Попова, 5