МИНОБРНАУКИ РОССИИ

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

БАЗОВЫЕ СВЕДЕНИЯ К ВЫПОЛНЕНИЮ КУРСОВОЙ И ЛАБОРАТОРНЫХ РАБОТ ПО ДИСЦИПЛИНЕ "ПРОГРАММИРОВАНИЕ". ПЕРВЫЙ СЕМЕСТР

Учебно-методическое пособие

Санкт-Петербург
Издательство СПбГЭТУ "ЛЭТИ"
2022

УДК 004.42(07)
ББК З 973.2–018я7
Б49

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

Б49 Базовые сведения к выполнению курсовой и лабораторных работ по дисциплине "Программирование". Первый семестр: учеб.-метод. пособие. СПб.: Изд-во СПбГЭТУ "ЛЭТИ", 2022. 84 с.

ISBN 978-5-7629-3166-3

Содержит материалы по темам дисциплины "Программирование" на языке Си. Предназначено для студентов 1 курса осеннего семестра обучения.

Составлено в форме краткого конспекта, содержащего как теоретический материал, так и поясняющие примеры. Для каждой работы приводятся вопросы для самоконтроля.

Предназначено для студентов направлений "Программная инженерия" и "Прикладная математика".

УДК 004.42(07)
ББК З 973.2–018я7

Рецензент канд. техн. наук Е. М. Линский (СПбГУАП).

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

ISBN 978-5-7629-3166-3

ВВЕДЕНИЕ

В пособии приводятся методические указания к лабораторным работам по дисциплине "Программирование" на языке Си для 1 курса осеннего семестра обучения. Перечень лабораторных работ соответствует рабочей программе дисциплины и включает:

1. Управляющие конструкции и отладка в языке Си.
2. Компиляция, сборка программ, автоматизация процесса сборки.
3. Использование указателей и массивы. Строки.
4. Операции со структурами. Обзор стандартной библиотеки.

Также пособие содержит указания к выполнению курсовой работы по теме "Обработка текстовых данных".

Для удобства изучения материал разделен на две большие части: общие базовые сведения и подробный разбор каждой лабораторной и курсовой работы. Более полное описание функций и возможностей стандартной библиотеки языка (libc) представлено в приложении.

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

Для обеспечения образовательного процесса по дисциплине используются следующие информационные технологии:

1. Операционная система: Ubuntu Desktop 20.04 x64.
2. Программное обеспечение общего и специализированного назначения: Git version 2.17, LibreOffice 6 и LibreOffice 6 Help Pack (Russian), GCC 7.5.0.
3. Информационные справочные системы: международная ассоциация сетей Интернет, электронные библиотечные системы и ресурсы удаленного доступа библиотеки СПбГЭТУ "ЛЭТИ".

Язык Си является высокоуровневым инструментом программиста и позволяет писать программы достаточно близкие к аппаратной части компьютера. Такие программы будет писать намного проще, если программист будет иметь общее представление об устройстве компьютера.

Основной компонент компьютера – **процессор**. Несмотря на то, что современные процессоры технически довольно сложны, их можно в некоторых случаях рассматривать как очень примитивные устройства. Однако чтобы изучить работу процессора, потребуется ввести еще одно устройство – **память**. В дан-

ном контексте речь идет об оперативной памяти компьютера. Ее можно представить себе как огромное количество подряд идущих ячеек размером в один байт. Все ячейки строго последовательно пронумерованы. Таким образом, каждая ячейка имеет адрес.

В памяти могут храниться как числа, так и инструкции процессора. Примечательно, что и данные, и машинные инструкции одинаковым образом могут быть сохранены в виде двоичных кодов в памяти.

Основной список операций, которые выполняет процессор:

1. Считать из памяти какую-то инструкцию (команду).
2. Считать из памяти данные, которые требуется для выполнения этой инструкции.
3. Выполнить инструкцию, используя считанные ранее данные.
4. Сохранить результат выполнения инструкции обратно в память.
5. Перейти к следующей инструкции.

Можно считать, что процесс программирования представляет собой написание такой последовательности инструкций, чтобы выполнение их подряд, одной за другой, приводило к решению поставленной задачи. Но так как писать программы в виде машинных инструкций очень неудобно и долго (равно как и отлаживать и поддерживать такие программы), программисты в основном пишут программы на языке, более удобном человеку, а потом преобразуют их в язык, понятный процессору. Это преобразование выполняет специальная программа, называемая компилятором.

1. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

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

```
#include <stdio.h>      // подключение библиотеки ввода/вывода

int main()              // определение функции main
{
  printf("Hello World!\n"); // вызов функции вывода в консоль
  return 0;       // окончание программы и возврат кода ошибки
}
```

Можно сохранить код этой программы в файл с любым удобным именем и расширением `.c`, например, `main.c`, можно скомпилировать её, выполнив команду:

```
gcc main.c
```

Результатом в операционной системе Линукс будет исполняемый файл с именем `a.out` в текущей директории. Исполняемые файлы **несовместимы** для компьютеров с различными архитектурами, а также для различных операционных систем. Поэтому эту программу нужно компилировать каждый раз под определенную архитектуру или операционную систему. Запустить исполняемый файл можно следующим 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 Си см. [1]–[3].

1.1. Переменные

Любая программа для выполнения своих целей совершает различные вычисления, результат которых меняется в зависимости от входных данных. Чтобы хранить различные данные, которые могут меняться в процессе работы программы, в языке Си существуют **переменные**.

Переменные, с точки зрения программиста, – это именованные области памяти, куда записывается результат вычислений.

Пример создания переменной:

```
int a;
```

Данной строчкой в программе была создана переменная с именем a. Объявление переменных имеет следующую структуру:

<Тип данных> имя_переменной;

В примере используется целочисленный тип **int**, т. е. данная переменная может принимать только целочисленные значения. Подробнее типы данных рассмотрены в теоретических положениях к лабораторной работе 1.

Значение данной переменной можно менять:

```
a = 10;
```

Стоит отметить, что обращаться и менять значение можно только после объявления переменной или во время объявления этой переменной, т. е. можно объединить объявление и присваивание числа 10 в одну строчку:

```
int a = 10;
```

Рекомендуется всегда присваивать переменной какое-то значение при объявлении (инициализировать переменную), так как если этого не сделать, то

значение этой переменной может быть любым, что может привести к непредвиденным ошибкам и ситуациям.

Важно: крайне желательно именовать переменные понятным образом, т. е. давать имя переменной в соответствии с целью, для которой она создавалась. В пособии рекомендуется именовать переменные символами нижнего регистра и через знак нижнего подчеркивания "_", если оно состоит из нескольких слов. Например, если целочисленная переменная была создана для хранения значения максимальной допустимой скорости на текущем участке дороги, то можно именовать переменную как `cur_max_speed` (сокращение от *current maximum speed*).

1.2. Функции

Чтобы лучше понять устройство программы на языке Си, рассмотрим базовое понятие "функция". Функция в языке Си – 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>
}
```

Пример:

```
void print(int a, int b){
    int c = a + b;
    printf("c = %d\n", c);
 }
```

где <тип_возвращаемого_значения> – `void`, имя_функции – `print`, <список_параметров_функции> задается в скобках – `int a, int b`, <Код_который_исполняется_в_функции> задается в фигурных скобках (тело 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. То есть в любом вызове функции и определении этой функции количество аргументов и типы этих аргументов должны совпадать. 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 <список_параметров_функции> никак не указывается.

Аргументы функции являются копиями данных, которые передаются в функцию, а все переменные, объявленные в теле функции, называются локальными переменными, т. е. переменные имеют область видимости в программе. Например, переменная может быть глобальной, если ее объявить в файле вне тела функции, и к ней можно получить доступ из любой функции программы в данном файле с исходным кодом. А к локальной переменной есть доступ только в теле функции, где она была объявлена, и только после ее объявления.

Выделяют четыре типа областей видимости:

– глобальная видимость (объявление в файле вне блоков/структур/функций);

– блок (область видимости в блоке, который начинается с фигурной скобки "{" и заканчивается "}");

– прототип функции (объявления внутри прототипа);

– функция (область видимости в теле функции).

Локальными переменными согласно данным типам областей видимости являются переменные, объявленные в блоке кода и в теле функции. Переменные, объявленные в прототипе функции, называются аргументами функции, а переменные в глобальной области видимости – глобальными переменными [4], [5].

1.3. Главная функция

Выполнение любой программы на языке Си начинается с выполнения функции **main** (говорят также, что main – точка входа в программу). Любая 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`. Можно переместить эту строку перед строкой с `printf` и убедиться, что операция вывода строки "Hello World! " на экране 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 и некорректное завершение программы.

1.4. 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 её 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.

1.5. Препроцессор

Препроцессор – это программа, которая подготавливает код программы для передачи ее компилятору. Она вызывается автоматически и вручную вызывать её не требуется. Команды препроцессора называются **директивами** и имеют следующий формат:

```
#ключевое_слово параметры
```

Основные действия, выполняемые препроцессором:

– удаление комментариев;

– включение содержимого файлов (`#include`);

– макроподстановка (`#define`);

– обработка директив условной компиляции (`#if, #ifdef, #elif, #else, #endif`).

1.5.1. #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.

1.5.2. #define

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 от **макроопределений** 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 `define` в [1].

1.5.3. #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.

1.5.4. Пример работы препроцессора

Рассмотрим работу препроцессора на примере программы:

```
// Пример работы препроцессора
#define B 14
#ifdef B
```

```
#define C
#endif

int main(){
   int a = 10;
   int b = B;
#ifdef C
   int c = b;
#endif
}
```

Команды препроцессора следующие: подставить значение 14 в места макроопределения `B`, затем, если существует макроопределение `B`, то подставить `C`. Если существует макроопределение `C`, то подставляется строка кода с переменной `c`.

Компилятор `gcc` с помощью флага `-E` позволяет запустить только программу препроцессора, чтобы можно было получить результат его работы:

```
gcc -E main.c -o prepr_main.c
```

Результат записанный в `prepr_main.c`:

```
int main(){
   int a = 10;
   int b = 14;
   int c = b;
}
```

Как можно заметить, сработали все макроподстановки и был удален комментарий.

2. КОМПИЛЯЦИЯ

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

Компилятор – программа, которая осуществляет компиляцию (рис. 2.1).

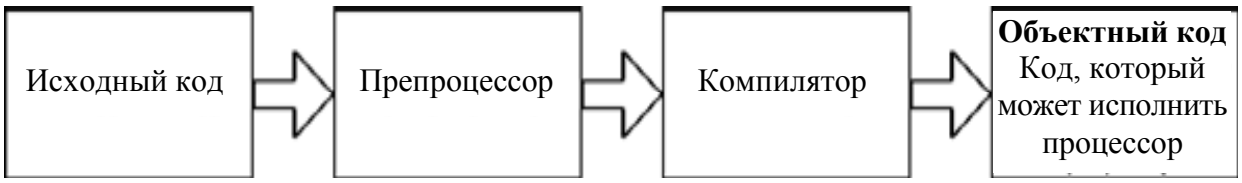


Рис. 2.1. Принцип работы компилятора

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 Си 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**.

Подробнее о компиляции в [4].

2.1. 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. 2.2.

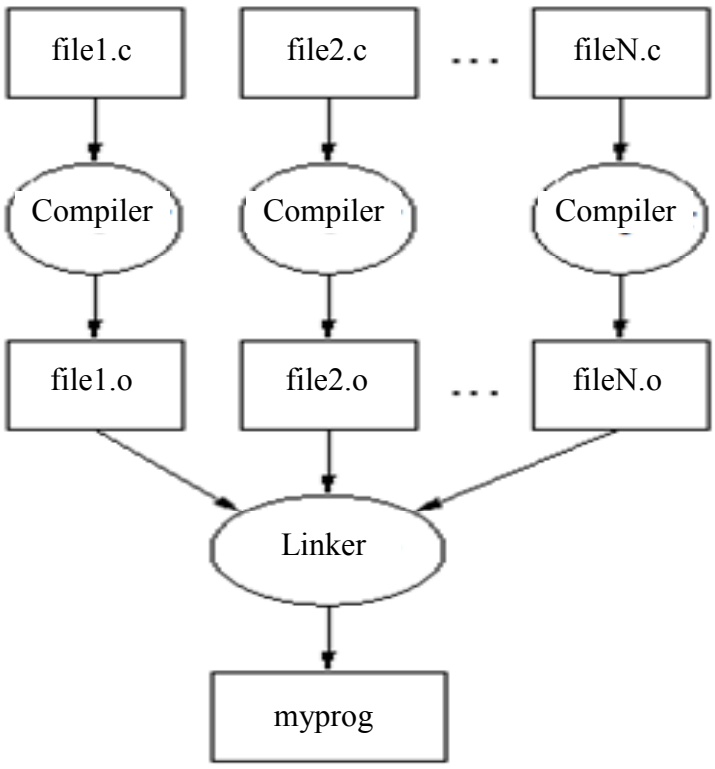


Рис. 2.2. Схема сборки программы из исходных кодов

Для получения объектного файла следует вызвать `gcc` с ключом `-c`:

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

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

Для линковки достаточно вызвать команду `gcc` с указанием объектных файлов. Чтобы изменить имя исполняемого файла программы используйте `-o <имя_файла>`. Если `-o` не указать, то по умолчанию исполняемым файлом является `a.out`. Пример изменения имени:

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

Подробнее о сборке программы в [6].

2.2. Перенаправление ввода/вывода

Зачастую пользователю необходимо передавать информацию в программу и получать результат из консоли. Для этого в ОС Линукс существуют **потоки ввода/вывода**. По своей сути **потоки** – это файлы, которые содержат передаваемую информацию, например, из программы в терминал или наоборот. В стандартной библиотеке языка Си существуют специальные функции, которые позволяют получать и передавать информацию в эти потоки.

Ввод и вывод распределяется между **тремя стандартными потоками**:

- `stdin` – стандартный ввод (ввод текста с клавиатуры). Считается потоком 0;
- `stdout` – стандартный вывод (вывод текста на экран). Считается потоком 1;
- `stderr` – стандартная ошибка (вывод ошибок на экран). Считается потоком 2.

Стандартный поток ввода передается в программу, когда пользователь вводит данные с помощью клавиатуры. Стандартный поток вывода и поток вывода ошибок отображаются в терминале в виде текста.

Можно перенаправить вывод программы с использованием специального символа >:

`./a.out > out.txt`, где `out.txt` – выходной файл. Если файл не существует, он будет создан, если существует – перезаписан.

Также можно перенаправить вывод в файл с добавлением в конец. Информация, хранящаяся в файле, не будет удалена, а вся новая информация будет добавлена в конец этого файла:

```
./a.out >> out.txt
```

Чтобы поместить стандартный вывод в файл:

```
./a.out < in.txt
```

И также можно использовать цифры, которыми задаются потоки вывода:

`./a.out 1>out.txt` – перенаправление вывода (`stdout`) в файл;

`./a.out 1>>out.txt` – перенаправление с добавлением в конец файла;

`./a.out 2>out.txt` – перенаправление ошибок (`stderr`) в файл;

`./a.out 2>>out.txt` – перенаправление с добавлением в конец файла;

`./a.out &>out.txt` – перенаправление и вывода (`stdout`), и ошибок (`stderr`) в файл .

3. ОТЛАДКА

3.1. Запуск с отладчиком gdb

`gdb` – это популярный **отладчик** для программ, написанных на языках программирования Си и C++. Он позволяет выполнять программу по шагам, а также посмотреть значения переменных на каждом этапе выполнения и, если это необходимо, рассмотреть программу даже на уровне машинных инструкций и регистров процессора.

Чтобы установить отладчик `gdb` в Ubuntu, введите:

```
sudo apt install gdb
```

Чтобы получить информацию об именах переменных и номерах строк кода во время отладки, программу следует скомпилировать особым образом, использовав ключ `-g`. Он добавит в исполняемый файл инструкции, которые будет использовать отладчик:

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

Чтобы запустить вашу программу с отладчиком, выполните:

```
gdb <имя программы>
```

Например:

```
gdb a.out
```

После запуска отладчика специальные символы будут считаны и запустится командный интерфейс отладчика. При этом программа не будет запущена, а запущен только отладчик, в котором можно управлять программой с помощью специальных команд:

- `run` или `r` – запустить программу;
- `run <arg1> <arg2>` – запуск программы, если необходимо передать аргументы;
- `where` – вывод стека вызовов, который привел к ошибке;
- `print <имя переменной>` или `p` – вывести значение переменной;
- `list` – просмотр исходного файла программы. По умолчанию выводит первые 10 строк;
- `help` или `h` – просмотр справки по команде;
- `quit` или `q` – выход из программы.

3.2. Перенаправление ввода/вывода

По умолчанию, программа, запущенная в gdb, осуществляет ввод и вывод в тот же терминал, что и gdb. Можно перенаправить ввод/вывод программы:

```
run > <выходной-файл>
```

Например:

```
run > 1.txt
```

3.3. Точки останова

С помощью **точек останова** можно остановить выполнение программы на любой строке или функции исходного кода. Это позволит проанализировать переменные и то, что происходит с программой.

С помощью команды `list` можно посмотреть исходный код программы и найти место точки вхождения в программу. На языке Си точкой вхождения является функция `main`. Также могут быть полезны команды

- `list <номер-строки>` – вывести строки, расположенные вокруг строки с номером <номер-строки> в текущем исходном файле;
- `list <имя функции>` – вывести строки, расположенные вокруг начала функции <имя функции>.

Можно установить точки останова различными способами:

- `break` или `b`
- `b <имя функции>`
- `b <номер строки>`

Также можно установить точку останова с условием:

- `break <номер строки> if <условие>`

Каждый раз, когда достигается точка останова, происходит вычисление выражения `<условие>`. Остановка происходит только, если эта величина не равна нулю, т. е. условие истинно.

Например, можно установить точку останова на вхождение в программу: `b main`. Затем запустить программу с помощью: `r`.

Чтобы выполнить следующую строку программы, нужно сделать шаг в `gdb`, выполнив

`next` или `n`.

GDB пропускает пустые строки и строки с комментариями, переходя к следующей строке. Если в строке `вызов функции`, то отладчик выполнит ее целиком, не заходя в функцию. Если нужно зайти в нее, то есть команда `шаг внутрь` (`step-in`):

• `step` или `s`;

• `info locals` позволяет узнать, какие локальные переменные сейчас инициализированы;

• `info breakpoints` – какие в данный момент есть точки останова;

• `del <breakpoint_num>` – удалить точку останова, где `<breakpoint_num>` – номер точки останова.

Данные о точках останова содержат следующую информацию:

• `Num` – номер точки останова;

• `Type` – точка останова, точка наблюдения или перехвата;

• `Disp` – помечена ли точка останова для отключения или удаления после активации;

• `Enb` – включенные точки останова помечаются как `y`, `n` отмечает выключенные точки;

• `Address` – адрес памяти, где расположена точка останова в программе;

• `What` – файл и номер строки, где расположена точка останова в исходном файле.

Также полезны функции:

• `continue` или `c` – перейти к следующей точке останова;

• `call <имя функции>` – вызывать локальные функции из программы;

• `finish` или `fin` – продолжить выполнение функции, но остановить программу, когда функция завершится. Использовать `finish` в главной функции нельзя;

• `kill` – завершить выполнение программы.

Подробнее об отладчике `gdb` в [7] и [8].

3.4. Пример отладки программы

Допустим, была написана программа:

```
#include <stdio.h>
int main()
{
   char *str = "GfG";
   /* Записываем адрес строки, доступной только для чтения */
   /* Проблема: попытка писать в память только для чтения */
   str[1] = 'n';
   return 0;
}
```

Скомпилируем и запустим ее:

```
$ ./a.out;
Ошибка сегментирования (стек памяти сброшен на диск).
```

Программа скомпилирована без ошибок, но при этом есть ошибка во время выполнения. По сообщению не понятно, где ошибка. Чтобы разобраться, можно скомпилировать программу с ключом `-g` и запустить её с помощью `gdb`:

```
$ gcc -g gdb2.c
$ gdb a.out
GNU gdb (Ubuntu 8.1.1-0ubuntu1) 8.1.1
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type
"show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from a.out...done.
```

Запуск программы:

```
(gdb) run
Starting program:a.out

Program received signal SIGSEGV, Segmentation fault.
0x0000555555554611 in main () at gdb2.c:10
10          str[1] = 'n';
```

Как было рассмотрено ранее, с помощью `gdb` можно увидеть, что ошибка в 10 строчке. Необходимо сначала остановить выполнение программы, а затем продолжить ее отладку, расставив точки останова. Код с выводом номеров строк:

```
(gdb) list
5       char *str;
6       /* Записываем адрес строки, доступной только для чтения */
7       str = "GfG";
8
```

```
9     /* Проблема: попытка писать в память только для чтения */
10         str[1] = 'n';
11         return 0;
12    }
```

Добавим точки останова на 7 и 10 строки:

```
(gdb) break 7
Breakpoint 1 at 0x5555555545fe: file gdb2.c, line 7.
(gdb) break 10
Breakpoint 2 at 0x555555554609: file gdb2.c, line 10.
```

Запустим программу снова:

```
(gdb) run
Breakpoint 1, main () at gdb2.c:7
7     str = "GfG";
```

Следующий шаг:

```
(gdb) next
Breakpoint 2, main () at gdb2.c:10
10         str[1] = 'n';
(gdb) next
Program received signal SIGSEGV, Segmentation fault.
0x0000555555554611 in main () at gdb2.c:10
10         str[1] = 'n';
```

Можно просмотреть данные о текущих точках останова:

```
(gdb) info breakpoints
Num   Type         Disp Enb Address          What
1      breakpoint  keep y  0x00005555555545fe in main at gdb2.c:7
       breakpoint already hit 1 time
2      breakpoint  keep y  0x0000555555554609 in main at gdb2.c:10
       breakpoint already hit 1 time
```

Останавливаем программу и gdb:

```
(gdb) kill
Kill the program being debugged? (y or n) y
(gdb) q
```

Затем можно внимательнее изучить код в строке 10, обнаружив ошибку. Она заключалась в том, что в переменной `str` хранится строковый литерал, доступный только для чтения, а в строчке 10 происходит попытка его модификации. Такая реализация приводит к аварийному завершению программы.

4. ИНТЕРЕСНЫЕ ФАКТЫ

4.1.

Оператор препроцессора ## применяется в сочетании с оператором #define. Оператор склеивания (или конкатенации) ## используется для объединения (конкатенации) двух лексем:

```
#include <stdio.h>
#define value(x) value_ ## x

int main()
{
   int value_a;
   value(a);

   return 0;
}
```

Препроцессор преобразует `value(a)` в `value_a`.

4.2. Порядок выполнения операций

В начале обучения программированию иногда могут возникать трудности связанные с **последовательностью выполнения** команд в различных ситуациях, например, префиксная и постфиксная форма инкремента (`++i` и `i++`) или при передаче аргумента в функцию. Рассмотрим ситуацию:

```
foo(i++, bar(i));
```

Порядок вычисления выражений, являющихся первым и вторым аргументом, не определен, поэтому функция **bar** может быть вызвана как до инкремента, так и после. Поэтому данный код не является корректным, так как не имеет однозначного поведения. Поведение в данной ситуации будет зависеть от компилятора, т. е. на различных компиляторах и даже на различных версиях компилятора может быть получен различный результат выполнения программы.

Лабораторная работа 1.
УПРАВЛЯЮЩИЕ КОНСТРУКЦИИ ЯЗЫКА СИ

1.1. Цель и задачи

Цель – освоение работы с управляющими конструкциями на языке Си 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.

1.2. Основные теоретические сведения

1.2.1. Типы данных и операции над ними

В языке Си имеется несколько основных типов данных:

• **char** – один байт. Обычно используется для хранения символов;

• **int** – целое число;

• **float** – вещественное число;

• **double** – вещественное число двойной точности.

Также есть ряд спецификаторов:

• **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 Си нет. Для хранения строк используются массивы типа `char`, в которых после последнего символа строки хранится нулевой символ `'\0'`.

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

В языке Си содержатся привычные арифметические операции: +, -, *, / и операция %. Дело в том, что операция деления (/) над целыми числами выполняется с отбрасыванием дробной 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, и как только становится ясным результат выражения, вычисление его аргументов прекращается.

Выражения в языке Си состоят из **операций** и **операндов**. **Операндами** являются данные, которые обрабатываются командами языка и **операциями**. Например, это переменные, числа и т. д.

В языке Си предусмотрены специальные операции увеличения – **инкремент** (++) и уменьшения – **декремент** (--). Операция инкремента прибавляет единицу к своему операнду, а декремент вычитает единицу.

Пример:

```
int n = k++;
```

Есть **префиксная** и **постфиксная** их версии. Отличие их в том, что префиксная операция выполняется до использования переменной в выражении `(++k)`, а постфиксная после `(k++)`. При этом операции инкремент и декремент можно применять только к переменным. Нельзя написать: `(n+k)++`. Про приоритет операций см. [1], разд. 2.12.

А что произойдет, если сложить два числа, которые по отдельности помещаются в тип данных, а их сумма – нет? Допустим, в ячейке памяти помещается 1 байт, что равно 8 битам. В 8 бит при беззнаковом представлении умещается число в интервале от 0 до 255 включительно. Что будет, если сложить 200+60? Число 260 больше 255, т. е. произошло переполнение, так как 260 не поместится в выделенную ячейку памяти для заданного размера. В итоге операции будет получен результат, равный 4, так как 260 соответствует следующему битовому представлению 100000100, это число представляется 9 битами, и старший (крайний левый) бит будет отброшен, а в результате останется 00000100, что соответствует числу 4.

Переполнение – это результат операции, при которой полученное значение не может быть помещено в конкретный тип данных. Более того, если сумма двух положительных знаковых целых чисел переполнится (станет больше, чем вмещает тип), результатом может иметь отрицательное значение. Переполнение может привести к серьезным ошибкам: потере данных, трудноуловимым ошибкам.

1.2.2. Приведение типов

Преобразование значения переменной одного типа в значение другого типа называется **приведением** типа. В языке Си существует **явное** и **неявное** приведение типов.

Явное приведение: перед выражением следует указать в круглых скобках имя типа, к которому необходимо преобразовать исходное значение. Пример:

```
int x = 5;
double y = 15.3;
x = (int) y;
y = (double) x;
```

Неявное приведение: преобразование происходит автоматически, по правилам, определенным в языке Си. Пример:

```
int x = 5;
double y = 15.3;
y = x; //здесь происходит неявное приведение типа к double;
x = y; //здесь происходит неявное приведение типа к int.
```

Рассмотрим другой пример, где вычисляется частное от деления:

```
int x = 15;
int y = 2;
float a = (float) x / y;
```

Круглые скобки используются, чтобы сообщить компилятору о необходимости преобразования переменной `x` (типа `int`) в тип `float`. Поскольку переменная `x` станет типа `float`, то `y` также затем автоматически преобразуется в тип `float`. Произойдет деление типа с плавающей точкой.

```
char x = 10;
char y = 20;
printf("Value is %d", x+y);
```

В результате выполнения такой операции будет выведено: `Value is 30`, так как сложение двух переменных типа `char` произведено как сложение двух типов `int`.

Отдельное внимание стоит уделить преобразованию типа `char` в `int`. Рассмотрим пример с неявным преобразованием:

```
char c = 10;
int i = c;
printf("Value is %d", i);
```

Переменной `i` сначала будет присвоено значение переменной `c`. У `c` тип имеет меньший диапазон, чему у `i`. Поэтому до присвоения произойдет неявное преобразование значения переменной `c` в тип `int`, а затем преобразованное значение будет присвоено переменной `c`. Преобразование значения типа, имеющего меньший диапазон, в тип, имеющий больший диапазон, является безопасным. В результате выполнения кода будет выведено: `Value is 10`.

Если попробовать выполнить обратное преобразование, то результат выполнения операции будет неправильным. Рассмотрим пример:

```
#include <stdio.h>
int main()
{
   int i = 1000000;
   char c = i;
   printf("Value is %d", c);
   return 0;
}
```

В результате выполнения кода, будет выведено: `Value is 64`. Число миллион не умещается в диапазон типа данных `char`, однако откуда взялось значение 64? Миллион в шестнадцатеричной системе имеет вид `0x000F4240`. Каждые два разряда этого числа – это один байт. Тип `char` является однобайтовым знаковым целым. При преобразовании от миллиона в шестнадцатеричной системе отбросятся 3 старших байта, и останется лишь один младший байт, а это и есть `0x40`. А в десятичной системе 64, что и было выведено. Такое преобразование является небезопасным.

Также необходимо быть внимательным при использовании неявного приведения типа при делении. Так как, если явно не указать, к какому типу нужно привести делимое и/или делитель, сначала выполняются вычисления, а потом приведение типов. Поэтому остаток от деления может быть потерян.

Преобразование при выполнении арифметических операций осуществляются проверкой следующих правил по порядку. Если предыдущее правило не выполняется, то проверка переходит к следующему:

- если один из операндов имеет тип `long double`, то другой приводится к `long double`;
- если один из операндов имеет тип `double`, то другой приводится к `double`;
- если один из операндов имеет тип `float`, то другой приводится к `float`;
- если один из операндов имеет тип `unsigned long int`, то другой преобразуется в `unsigned long int`;
- если один из операндов имеет тип `long int`, а другой – `unsigned int`, то:

– если `long int` покрывает все значения `unsigned int`, то результат будет типа `long int`;

– если нет, то оба операнда преобразуются в `unsigned long int`;

- если один из операндов имеет тип `long int`, то другой приводится к `long int`;

• если один из операндов `unsigned int`, то другой приводится к `unsigned int`;

• в противном случае оба операнда имеют тип `int`.

Как было рассмотрено ранее, иногда преобразования сопровождаются потерей информации. Без потери информации можно выполнить следующие цепочки преобразований:

```
char -> short -> int -> long;
unsigned char -> unsigned short -> unsigned int -> unsigned long;
float -> double -> long double.
```

Исходя из размерности каждого типа данных ясно, что типы данных большой размерности вмещают больший интервал значений. Такие типы данных будем называть **более точными**, а меньшей размерности – **менее точными** типами. На рис. 1.3 представлена градация типов данных по точности, из которого следует, что самым точным типом в языке Си является `long double`.

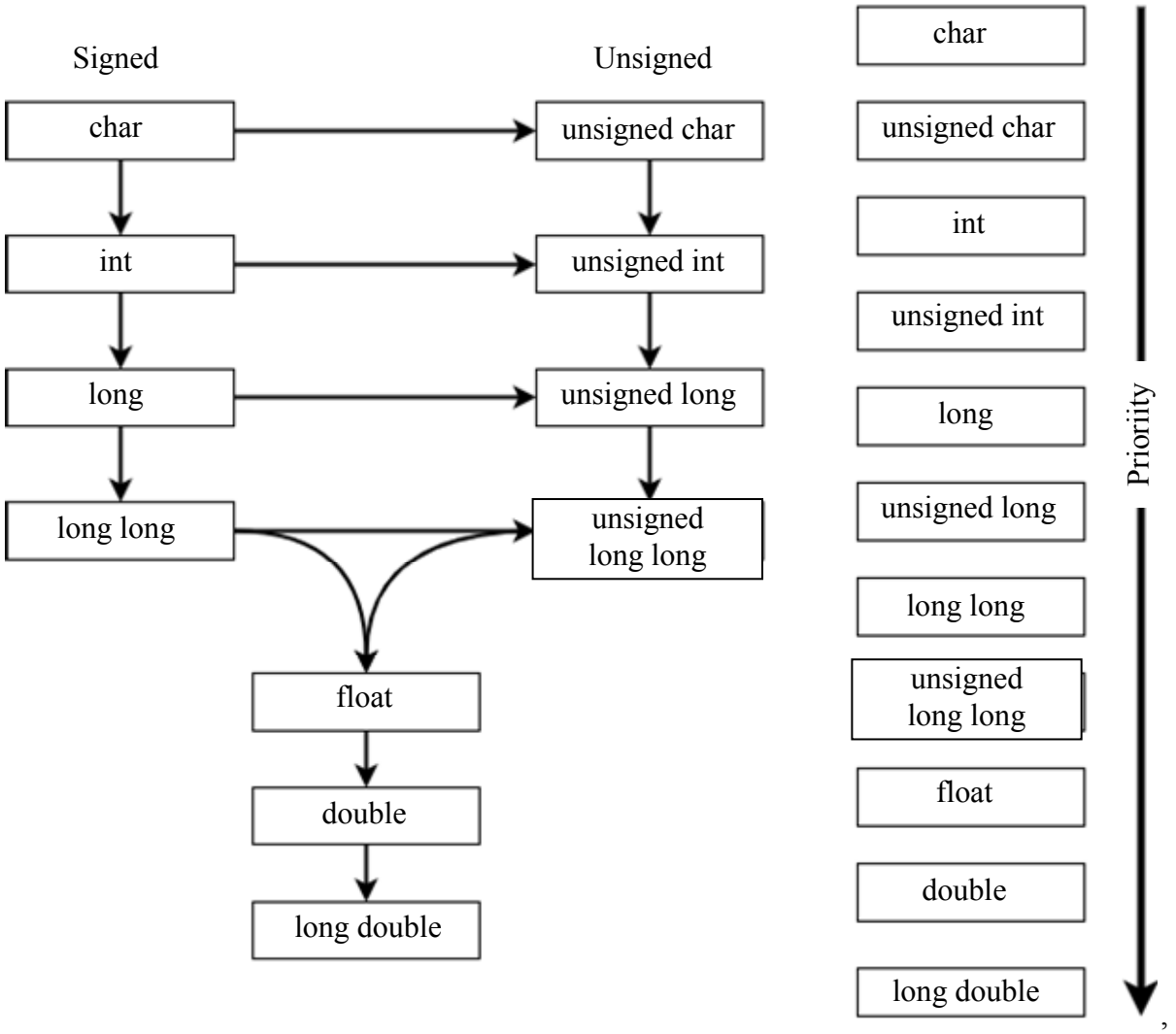


Рис. 1.3. Схема преобразования типов в выражении

Автоматически производятся только преобразования к **более точному** типу в соответствии с типом на рис. 1.3. Преобразование **к менее точному типу** происходит с потерей данных. Например, если в программе есть строчка: `int a = 4.9`, тип `int` относится к целым числам, а число, которое мы хотим присвоить, 4.9 – число с плавающей точкой, поэтому произойдет преобразование к типу `int`. При преобразовании число не округляется, дробная часть будет отброшена, поэтому в итоге `a = 4`.

Потеря точности возникает в следующих ситуациях:

– передача значения как аргумент функции, где аргумент имеет тип меньшей точности;

– присваивание;

– явное преобразование типов.

Подробнее про преобразования см. [9].

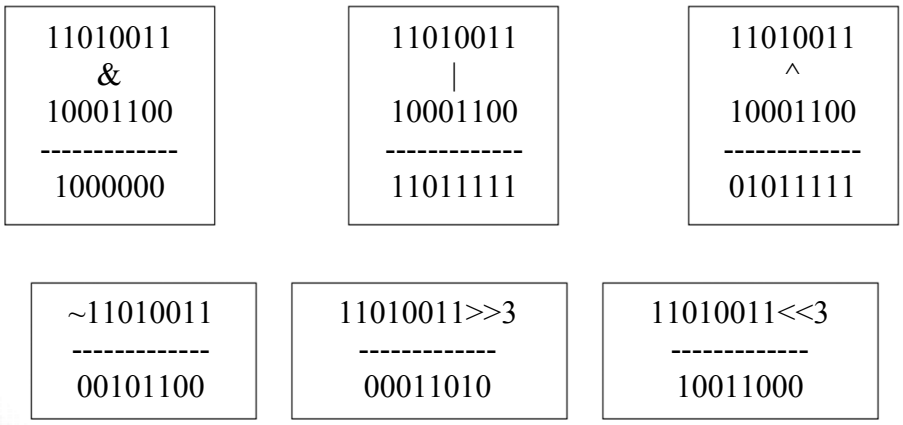
1.2.3. Поразрядные (побитовые) операции

В языке Си 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).

При выполнении поразрядных операций И, ИЛИ, исключающее ИЛИ для двух чисел каждой пары разрядов этих чисел независимо выполняется одна из этих операций. Дополнение ~ – операция только над одним аргументом. Она обращает каждый единичный разряд в нулевой, и наоборот:

a	**b**	**a&b**	**a|b**	**a^b**	**~a**
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0



Операция отрицания также инвертирует каждый отдельно взятый бит числа. Операции << и >> выполняют сдвиг всех разрядов числа на некоторое количество разрядов. Количество разрядов обязательно должно быть неотрицательным. "Вытолкнутые" разряды (например, при сдвиге вправо) теряются. Места сдвинутых разрядов (при сдвиге влево) заполняются нулями. Подумайте, на какую математическую операцию похожи сдвиги.

Операции И (&) и ИЛИ (|) следует отличать от логических операций && и ||, которые при вычислении слева направо дают значение истинности. Например, если x = 1, а y = 2, то x & y = 0, а x && y = 1.

Подробнее об этих операциях можно см. [1], разд. 2.9.

1.2.4. Основные конструкции в языке Си

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 <оператор 1> выполняется. Также оператор `if` может иметь необязательную ветку `else`, которая будет выполнена, 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. В случае, 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`.

Пример:

```
switch (num) {
   case 1:
       printf("Один");
       break;
   case 2:
       printf("Два");
       break;
   default:
       printf("Не цифра");
       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 два блока: начальное выражение, выполняемое один раз перед первым началом цикла, и выражение приращения, выполняемое после каждой итерации цикла.

В некоторых ситуациях необходимо прерывать выполнение цикла или перейти на следующую итерацию до завершения всего тела цикла. Для таких ситуаций существуют операторы `break` и `continue`.

Оператор `break` – досрочно прерывает выполнение цикла.

Пример:

```
int a = 2;
for(int i=0; i<10; i++){
   if(i % 2 == 0)
       break;
   a = a * a;
}
```

В данном примере цикл заканчивается, если остаток от деления итератора на два равен нулю. Таким образом переменная `a` ни разу не будет возведена в квадрат.

Оператор `continue` – досрочный переход к следующей итерации цикла.

Пример:

```
for(int i=0; i<10; i++){
   if(i % 2 == 0)
       continue;
   a = a * a;
}
```

Пример аналогичен предыдущему, но в данном случае переменная `a` будет возводиться в квадрат только при нечетном значении переменной `i`.

Подробнее про циклы в [10].

1.2.5. Функция форматного вывода printf

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

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

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

1.2.6. Функция форматного ввода scanf

```
int scanf ( const char * format, &arg1, &arg2, …&argN);
```

читает символы из стандартного ввода, интерпретирует их в соответствии с форматом, указанном в аргументе `format`, и помещает результаты в остальные

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

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

Функция возвращает количество успешно считанных аргументов; подробнее о символах управляющей строки см. [1], разд. 7.4.

При использовании `scanf` нужно указывать символ & перед именем переменной, иначе компилятор выдаст предупреждение, а программа завершится аварийно. Почему именно так происходит, и что обозначает в данном контексте & будет рассмотрено в лабораторной работе 3.

Пример:

```
#include <stdio.h>
int main()
{
   int a, b, c;
   c = scanf("%d %d", &a, &b);
   printf("Number of successful "
          "inputs read : %d",
          c);
   return 0;
}
```

Уже можно заметить, что при использовании функций `printf`, `scanf`, необходимо указывать спецификаторы. Рассмотрим наиболее часто используемые спецификаторы, которые понадобятся при выполнении лабораторных работ:

- %c – считать/вывести один символ;
- %d – считать/вывести десятичное число целого типа;
- %i – считать/вывести десятичное число целого типа;
- %f – считать/вывести число с плавающей запятой;
- %s – считать/вывести строку;
- %x – считать/вывести шестнадцатиричное число;
- %p – считать/вывести адрес указателя.

Строка формата считывается слева направо, при этом устанавливается соответствие между кодами формата и аргументами из списка аргументов. Подробнее про спецификаторы можно узнать в справочнике [11].

1.2.7. Отладка с помощью логов

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

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

здесь __FILE__ , __func__ , __LINE__ – специальные **макросы**, которые заменяются препроцессором на имя, полный путь к файлу, название функции и номер строки соответственно.

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 произойдет сбой, то можно будет по ее выводу определить, на какой именно строке он произошел. Подробнее о данных макросах можно прочитать в [12], [13].

1.2.8. 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" . Эту команду необходимо использовать после любых манипуляций с цветом, 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 [14] 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()
{
   printf("%sHello, %sworld!%s\n", FIRST, SECOND, NONE);
   return 0;
}
```

Дополнительные материалы см. [14], [15]. Также больше примеров содержится в [16] и [17].

1.3. Задание к лабораторной работе 1

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

В зависимости от значения программа должна выводить следующее:

- 0 : найти количество элементов до первого элемента, кратного 10 (`count_before_first_element`);
- 1 : индекс последнего элемента, кратного 10 (`index_last_element`);
- иначе необходимо вывести строку "Данные некорректны".

1.3.1. Описание последовательности выполнения работы

Сначала реализуем функцию для условия 0. Функция принимает на вход два аргумента: массив `arr[]` и кол-во элементов в массиве `count`. Инициализируем переменную для записи результата `result` нулем. Это важно, так как если не написать `= 0`, в переменную может быть записан мусор. Попробуйте сами запустить программу без присваивания 0.

Затем нужно написать цикл, который проходит по всем элементам в массиве. Например, в примере используется `for`.

Далее проверяется условие: остаток от деления элемента массива на 10 должен быть равен 0, иначе говоря, кратен ли элемент 10. Если да, нужно записать в результат количество элементов, предшествующих найденному.

На этом шаге уже известен индекс элемента, кратного 10. В примере индекс элементов начинается с 0, значит, для вычисления результата надо прибавить 1 к индексу элемента, кратного 10. Но при этом в условии сказано, что элемент, кратный 10, учитывать не нужно, нам нужны числа до этого элемента, поэтому из результата надо также вычесть 1. Итого, получается `result = i + 1 −1 = i`.

И так как в условии сказано, что нужно найти только первый элемент, следует прервать выполнение цикла, как только этот элемент будет найден. Поэтому используется `break`.

Функция возвращает результат выполнения операции:

```
int count_before_first_element(int arr[],int count) {
   int result = 0;

   for (int i = 0; i < count; i++) {
       if ((arr[i] % 10) == 0) {
           result = i;
```

```
            break;
        }
    }
    return result;
}
```

Аналогично реализуем функцию для нахождения индекса последнего элемента, кратного 10. Большая часть кода будет одинаковой, так как нужно тоже сначала найти элемент, кратный 10. Ранее в функции мы прерывали выполнение цикла, так как нужно было найти только первый элемент, здесь нужен последний элемент, поэтому можно удалить `break`, и в `result` будет записан последний элемент, кратный 10. В результате функция должна вернуть индекс, так что пишем `result = i`. Возвращаем результат:

```
int index_last_element(int arr[],int count) {
    int result = 0;
    for (int i = 0; i < count; i++) {
        if ((arr[i]%10) == 0) {
            result = i;
        }
    }
    return result;
}
```

Осталось подключить нужные библиотеки, задать максимальный размер массива N, реализовать функцию `main()`. Ввод массива с клавиатуры и подсчет количества элементов массива реализован в цикле `while`. Оператор `switch-case` используется для вызовов функции в зависимости от введенного пользователем 0 или 1:

```
#include <stdio.h>
#define N 20

int main()
{
    int k=0;
    int arr[N];
    int i=0;
    char space =' ';
    scanf("%d", &k);
    while (i < N && space == ' ') {
        scanf("%d%c", &arr[i], &space);
        i++;
    }
```

```
    switch(k) {
        case 0:
            printf("%d\n", count_before_first_element(arr,i));
            break;
        case 1:
            printf("%d\n", index_last_element(arr, i));
            break;
        default:
            printf("Данные некорректны");
    }
    return 0;
}
```

1.3.2. Пример выполнения задания на защиту

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; строку `false` в противном случае.

Код на языке Си:

```
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");
}
```

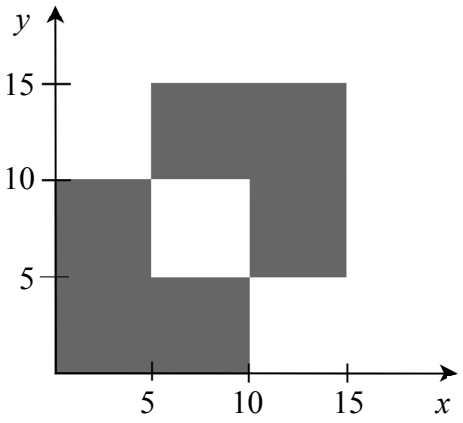


Рис. 1.4. Пример области

Полезно знать: код данной программы проверяет ряд случаев и, по сути, делит закрашенную область на два квадрата, вырезая четверть квадрата, которая пересекается по диагонали (рис. 1.4). Для качественного тестирования этой программы необходимо проверить ее работу на различных данных.

В данном случае для **качественного тестирования** необходимо проверить работу программы в точках на всех границах области. Границами области являются ребра двух многоугольников. Таким образом для каждого ребра этих многоугольников можно составить три точки: вне области, но рядом с ребром, на ребре и внутри области рядом с ребром.

Такой набор входных данных позволит покрыть наибольшее количество ситуаций для данной задачи, а значит, вне зависимости от реализации самой программы можно будет оценить корректность ее работы.

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 2.
СБОРКА ПРОЕКТОВ В ЯЗЫКЕ СИ

2.1. Цель и задачи

Цель работы – изучение процесса сборки программ, написанных на языке Си на примере использования make-файлов.

Для достижения поставленной цели требуется:

- изучить протекание процесса компиляции и линковки с использованием компилятора `gcc`;
- изучить структуру и правила составления `make`-файлов;
- написать `make`-файл для сборки заданной программы.

2.2. Основные теоретические сведения

2.2.1. 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, если это не требуется. Например, в случаях, когда в большом проекте изменился только один файл, утилита скомпилирует только его и не станет перекомпилировать все остальные. 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 описанная в файле цель. С нее и начинается обработка файла утилитой make.

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.

Нужно обязательно включать заголовочные файлы в make-файл. Если заголовочный файл будет изменен, а исполняемый – нет, то перекомпиляция и

сборка будут исполнены только для тех файлов, которые указаны в make-файле. То есть может быть ситуация, что объектный файл не изменится, поэтому при проверке зависимостей компилятор не узнает, что заголовочный файл был изменен. А если заголовочный файл указать в make-файле, то компилятор проверит его в зависимостях.

Допустим, была написана программа, которая складывает два числа и состоит из нескольких файлов.

print.h

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

print.c

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

main.c

```
#include <stdio.h>
#include "print.h"
int main(){
   int a = 10;
   int b = 20;
   int c = print(a, b);
   printf("%d", c);
   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
```

После запуска программы, будет выведен ответ 30. Если потребуется поменять файл print.h, а затем заново запустить сборку программы: make -f Makefile, будет выведено:

```
make: "example" не требует обновления.
```

Однако файл print.h был изменен. Таким образом, необходимо было указать в зависимостях и заголовочный файл:

makefile

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

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

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

При таком `make`-файле программа будет перекомпилирована корректно.

Подробнее про организацию файлов в языке Си см. [18].

2.2.2. Вывод строки в консоль, копирование и конкатенация строк

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);
```

Функция `puts` выводит строку типа `char` и символ новой строки в стандартный поток вывода, возвращает `EOF` в случае ошибки или неотрицательное значение, если запись прошла нормально:

```
#include <stdio.h>
int main()
{
   char s[100] = "Привет"; //Вывод строки
   puts(s);
   return 0;
}
```

Прототип функции копирования строк:

```
char * strcpy( char * destptr, const char * srcptr );
```

Функция копирует строку `srcptr`, включая завершающий нулевой символ, в строку назначения, на которую ссылается указатель `destptr`. Чтобы избежать переполнения, строка, на которую указывает `destptr`, должна быть достаточно длинной, чтобы в неё поместилась копируемая строка, включая завершающий нулевой символ.

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 количе-

ство копируемых символов num, то после скопированных символов неявно добавляется символ конца строки. Функция возвращает указатель на строку с результатом конкатенации.

Пример:

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

int main()
{
   char str1[40];
   char str2[40];

   strcpy (str1,"Быть ");
   // скопировать строку "Быть" в str1
   strcpy (str2,"или не быть");
   // скопировать "или не" и добавить к строке str2
   strncat (str1, str2, 11); // объединить строки str1 и str2
   printf("%s", str1);
   return 0;
}
```

Данный код программы выведет на экран строку: "Быть или не быть".

2.3. Задание к лабораторной работе 2

В этой работе требуется правильно вынести каждую функцию первой лабораторной в отдельный файл и написать Makefile для получившегося проекта.

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.

2.3.1. 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.

Аналогичное задание, как в первой лабораторной работе, но необходимо добавить Makefile. Например, написать программу с Makefile, которая считает и выведет на экран среднее значение в массиве целых чисел.

2.3.2. Пример выполнения задания на защиту

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;
}
```

`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
```

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 3.
ИСПОЛЬЗОВАНИЕ УКАЗАТЕЛЕЙ

3.1. Цель и задачи

Цель работы – освоение работы с указателями и динамической памятью.

Для достижения поставленной цели требуется:

- 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

3.2.1. Указатели

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

* – оператор разыменования;

& – оператор взятия адреса.

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

```
int *p;      // объявляем указатель;
int a = 10; // объявляем и инициализируем переменную;
p = &a; // в переменную-указатель записываем адрес переменной a;
*p = 25; // обратиться к ячейке по адресу, который хранится в p.
```

Обратим внимание, что в первой строке знак "*" относится к типу данных, т. е. типом является `int *`, а в последней строке примера знак "*" является оператором разыменования. На рис. 3.1 представлена иллюстрация указателя в языке Си согласно приведенному примеру.

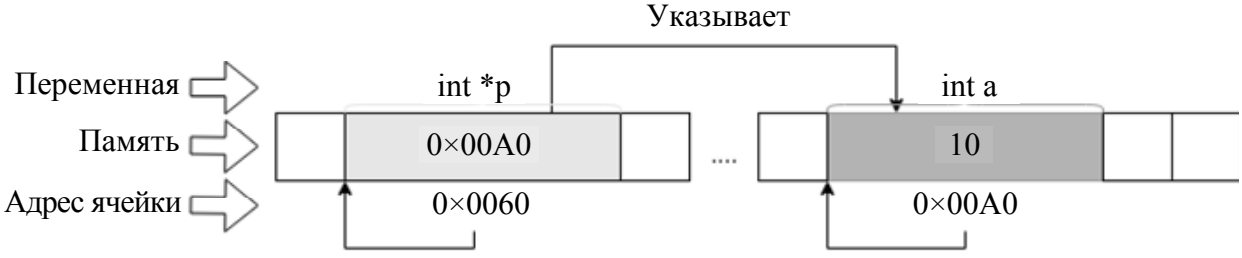


Рис. 3.1. Иллюстрация указателей в Си

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

Указатели хранят адреса объекта в памяти компьютера. Для получения адреса переменной используется операция `&a`. Эта операция применяется только к переменным и элементам массива.

Также стоит отметить, что должно соблюдаться соответствие типов. То есть, если переменная a имеет тип `int`, то и указатель, который указывает на ее адрес, тоже будет иметь тип `int`. При этом указатель на `void` может указывать на объекты любого типа. Для вывода значения указателя с помощью функции `print` можно использовать спецификатор `%p`:

```
#include <stdio.h>

int main(void)
{
   int *p;
   int a = 10;
   p = &a;
   printf("Address = %p \n", p);      // вывод адреса
   return 0;
}
```

В результате работы программы будет выведено число в шестнадцатеричном формате. Это число является адресом, по которому располагается переменная `a`. В зависимости от архитектуры, указатель может иметь разный размер. Наиболее вероятно, что в вашем случае размер будет 8 байт.

Так как указатель хранит адрес, то можно также по этому адресу получить хранящееся там значение, т. е. значение переменной `a`. Для этого применяется операция разыменования. Результатом этой операции всегда является объект, на который выведет указатель. В результате выполнения этой операции будет выведено значение переменной `a`:

```
#include <stdio.h>

int main(void)
{
   int *p;
   int a = 10;
   p = &a;
   printf("Address = %p \n", p);
   printf("a = %d \n", *p);
   // разыменование указателя p в аргументе функции printf
   return 0;
}
```

Либо можно присвоить значение, полученное в результате операции разыменования, другой переменной:

```
int a = 10;
int *p = &a;
int y = *p;
printf("a = %d \n", y); // 10
```

Чтобы поменять значение по адресу, который хранится в указателе:

```
int a = 10;
int *p = &a;
*p = 45;
printf("a = %d \n", a); // 45
```

Так как по адресу указателя располагается переменная `a`, то, соответственно, ее значение изменится.

Подробнее про операторы см. [5], [9], [10].

3.2.2. Массивы в языке Си

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

Пусть 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 и того же массива, то к ним можно применять операторы отношения ==, !=, <, >= и т. д. Например, отношение вида `p < q` истинно, если `p` указывает на элемент массива с меньшим индексом, 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. Подробнее арифметика указателей будет рассмотрена далее.

3.2.3. Арифметика указателей

Указатели поддерживают арифметические операции.

Могут быть применены сложение, вычитание, инкремент, декремент. Операция +2 сдвигает указатель вперед на `2*sizeof(тип)` байт. Так как операции выполняются с указателями, то вычисления выполняются над числами в шестнадцатиричном формате. Например, если указатель

```
int *a;
```

хранит адрес 120, то после

```
a += 2;
```

он будет хранить адрес 120 + `sizeof(int)` *2 = 120 + 8 = 128, так как в большинстве 64-разрядных систем `int` == 4 байта, но в зависимости от системы значение может отличаться.

Рассмотрим пример с указателем на начало массива:

```
#include <stdio.h>
void main() {
   int A[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   int *p;
   p = A;
   printf("%d\n", *p); //Вывод: 1 – первый элемент
   p++;
   printf("%d\n", *p); //Вывод: 2 – сдвинулись на 1 вправо
                          от первого элемента
   p = p + 4;
   printf("%d\n", *p); //Вывод: 6 – сдвинулись на 4 вправо
                          от второго элемента
}
```

Чтобы получить адрес первого элемента массива можно не использовать оператор &:

```
p = A;
```

Либо можно было бы написать:

```
p = &A[0];
```

Помимо + и – указатели поддерживают операции сравнения. Если есть два указателя `p` и `q`, то `p` > `q`, если адрес, который хранит `p`, больше адреса, который хранит `q`.

```
#include <stdio.h>
#define N 10
void main() {
```

```
    int A[N] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int *p, *q;

    p = &A[0];
    q = &A[N-1];          // sizeof(A)/sizeof(int)-1

    printf("&A[0] == %p\n", p); // &A[0] == 0x7ffde4250610
    printf("&A[9] == %p\n", q); // &A[9] == 0x7ffde4250634

    if (p < q) {
        printf("p < q");    // будет выведено p < q
    } else {
        printf("q < p");
    }
}
```

Если указатели равны, то они указывают на одну и ту же область памяти.

При работе с указателями надо отличать операции с самим указателем и операции со значением по адресу, на который указывает указатель:

```
int a = 10;
int *pa = &a;
int b = *pa + 20;    // операция со значением, на который
                        указывает указатель
pa++;                // операция с самим указателем

printf("b=%d \n", b);    // b=30
```

Через операцию разыменования `*pa` получено значение, на которое указывает указатель `pa`, т. е. число 10. Затем выполняется операция сложения.

Необходимо особое внимание уделить приоритетам операции, так как *, ++ и -- имеют одинаковый приоритет и при размещении рядом выполняются справа налево. Например:

```
int a = 10;
int *pa = &a;
printf("pa: address=%p \t value=%d \n", pa, *pa);
// pa: address=0x7fff8de09168   value=10
int b = *pa++;   // инкремент адреса указателя

printf("b: value=%d \n", b);   //b: value=10
printf("pa: address=%p \t value=%d \n", pa, *pa);
// pa: address=0x7fff8de0916c   value=10
```

В выражении:

```
b = *pa++;
```

сначала к указателю прибавляется единица, т. е. к адресу добавляется 4, так как указатель типа `int`. Так как инкремент постфиксный, с помощью операции

разыменования возвращается значение, которое было до инкремента, т. е. указатель на область памяти, где хранится значение 10. И это число 10 присваивается переменной b.

Ситуацию можно изменить следующим образом:

```
b = (*pa)++;
```

Скобки изменяют порядок операций. В этом случае сначала выполняется операция разыменования, получение значения, а затем это значение увеличивается на 1. По адресу в указателе будет находиться число 11. Так как инкремент постфиксный, переменная b получает значение, которое было до инкремента, т. е. число 10. Таким образом, в отличие от предыдущего случая все операции выполняются над значением по адресу, который хранит указатель, но не над самим указателем.

Аналогично и с префиксным инкрементом:

```
b = ++*pa;
```

Сначала с помощью операции разыменования берется значение по адресу из указателя pa, к этому значению прибавляется единица, то есть теперь значение по адресу, который хранится в указателе, равно 11. Затем результат операции присваивается переменной b.

Рассмотрим еще один случай:

```
b = *++pa;
```

Сначала изменяется адрес в указателе, затем берется значение по этому адресу и присваивается переменной b. Полученное значение в этом случае может быть неопределенным.

3.2.4. Передача аргумента в функцию

До сих пор мы писали свои функции, в которые передавали аргументы по значению. Это значит, что если аргумент передан в функцию и изменен там, в вызывающей функции изменения не сохранятся. Это происходит, поскольку функция получает **копию** аргумента и не может повлиять на его оригинал. Копирование данных в языке Си происходит каждый раз, когда передается значение из одной функции в другую.

Рассмотрим функцию swap, которая по замыслу должна менять значения целых переменных:

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

Как уже известно, в данном случае функция `swap` получает копии аргументов `a` и `b`, что называется передачей по **значению**, поэтому, если вызвать её 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;
}
```

Чтобы функция `swap` могла модифицировать свои аргументы, несколько ее изменим:

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

Теперь она принимает два указателя на переменные типа `int` и модифицирует значения, на которые они ссылаются. В данном примере копируется адрес переменной `a` и `b`, что позволяет менять значения этих переменных в любой точке программы, где известны их адреса. Такая передача данных из функции в функцию в языке Си называется передачей по **ссылке** (по указателю).

Важно отметить, что в других языка программирования, например, таких как Python, C++ или Java, термин "ссылка" имеет особое значение, не связанное с указателями. Это нужно не забывать, используя данную терминологию.

Передача данных по ссылке (указателю) может происходит быстрее в случаях, когда размер указателя занимает меньше памяти, чем данные по адресу, которые там хранятся. Это происходит потому, что необходимо скопировать меньше данных.

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 на выделенную память указатель, который следует привести к требуемому типу.

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

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

Если необходимо инициализировать массив нулями, то можно использовать функцию `calloc`:

```
void *calloc(size_t num, size_t size);
```

В отличие от `malloc` аргументами данной функции являются количество элементов массива `num` и размер элемента `size`, из которых рассчитывается общий объем необходимой памяти.

Также стоит отметить, что любая из перечисленных ранее функций может вернуть NULL, если возникает ошибка, например, у системы больше нет свободной памяти. Это обстоятельство всегда нужно проверять перед работой с указателем, куда записывается адрес выделенной памяти. В случае, если передать размер выделяемой памяти равным 0, то поведение функции выделения памяти не определено. Она получает указатель на выделенную ранее область памяти и новый 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). Следует понимать, что функция `realloc` может выполняться в некоторых случаях довольно долго, поэтому не следует использовать ее слишком часто без явной на то необходимости.

После того, как выделенная память больше не требуется, следует обязательно высвобождать ее с помощью функции `free`. Пример:

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

Освобождение памяти необходимо, чтобы избежать ее утечек. Утечкой памяти называют ситуацию, когда программа теряет указатель на участок памяти, что не позволяет получить этот участок повторно. Рассмотрим пример:

```
void foo(){
   int *p = (int *) malloc(5 * sizeof(int));
}
int main(){
   foo();
   …
   return 0;
}
```

В данной программе после вызова функции `foo()` происходит утечка памяти, так как доступа к переменной p уже нет, а значит и нет адреса выделенной памяти. Вызов этой функции в бесконечном цикле приведет к аварийному завершению программы из-за нехватки памяти. Утечки памяти влияют как на саму программу, так и на все программы, запущенные в системе, так как память становится недоступной для всех.

Также освобождение памяти для адреса, который не принадлежит программе или уже освобожден, приведет к неопределенному поведению. Для корректного освобождения памяти с помощью функции `free` необходимо передать ей аргумент, содержащий адрес начала участка памяти или NULL. Остальные ситуации приводят к неопределенному поведению.

Пример некорректного освобождения памяти:

```
int *p = (int *) malloc(5 * sizeof(int));
free(p+1);
free(1200);
free(p+10);
```

3.2.5. Строки в языке Си

Формально в языке Си нет специального типа данных для строк, но представление их довольно естественно: строки в языке Си – это массивы символов, завершающиеся нулевым символом (`'\0'`). Это порождает 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 исполь-
зуется 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 уже можно догадаться, если строка в Си – массив символов, то массив строк – это двумерный массив символов, где каждая строка – массив, хранящий очередную символьную строку.

3.2.6. Двумерные массивы

В предыдущем разделе были рассмотрены одномерные массивы, которые позволяют хранить множество элементов определенного типа. Указатели в языке Си позволяют указывать не только на обычные структуры и типы данных, но и на указатели. Таким образом появляется возможность создавать **многомерные** массивы. Рассмотрим теперь двумерные массивы как частный случай многомерных массивов.

Двумерный статический массив можно объявить следующим образом:

```
int arr[2][3];
```

Статические многомерные массивы в памяти представляют из себя непрерывную последовательность массивов (рис. 3.2), в то время как динамический массив представляет собой набор массивов, которые расположены в различных участках памяти.

arr[0][0]	arr[0][1]	arr[0][2]	arr[1][0]	arr[1][1]	arr[1][2]

Рис. 3.2. Представление двумерного статического массива массива в памяти компьютера

Объявление динамического двумерного массива происходит чуть сложнее. Рассмотрим пример создания массива размера 5×5:

```
int **arr = malloc(5*sizeof(int *));
for(int i=0; i<5; i++)
   arr[i] = malloc(5*sizeof(int));
```

Динамический двумерный массив является массивом указателей на массивы элементов, которые хранят соответствующие данные (рис. 3.3).

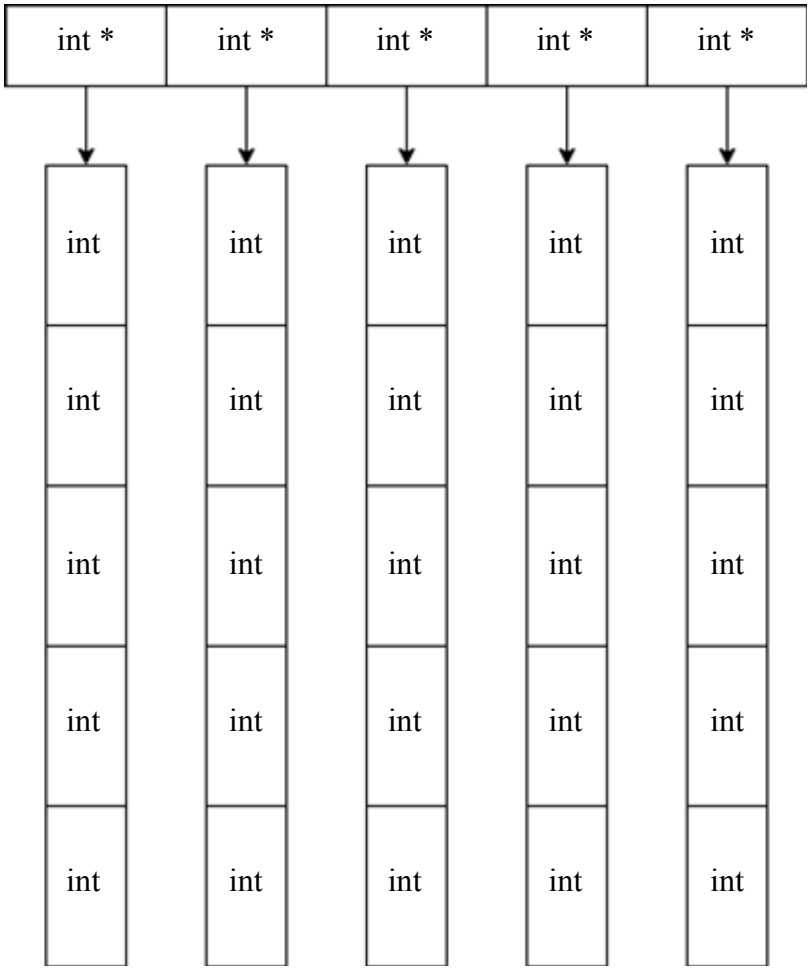


Рис. 3.3. Схема представления двумерного динамического массива в памяти компьютера

Обращаться к элементом массива можно также, как в одномерном случае, либо индексацией:

```
int el = arr[1][2];
```

или арифметикой указателей:

```
int el = *(*(arr+1)+2);
```

3.3. Задание лабораторной работе 3

Напишите программу, которая считывает текст до символа `"@"` и удаляет все слова, состоящие только из букв верхнего регистра. Предложения заканчиваются точкой, разделитель между предложениями – пробел.

Программа должна изменить и вывести текст следующим образом:

- каждое предложение должно начинаться с новой строки;
- табуляции, пробелы, перенос строки в начале предложения должны быть удалены;
- текст должен заканчиваться фразой "Количество букв в верхнем регистре до: `n` и количество букв после: `m`", где `n` – количество символов верхнего

регистра в изначальном тексте и `m` – количество символов верхнего регистра в отформатированном тексте.

Порядок предложений не должен меняться. Статически выделять память под текст нельзя. Пробел между предложениями является разделителем, а не частью какого-то предложения.

3.3.1. Описание последовательности выполнения работы

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

В первую очередь в программе необходимо задать константу для размера размера текста

```
#define TEXT_SIZE 100
```

Затем напишем функцию для чтения текста до терминального символа `"@"`:

```
int readText(char **text){
   int size = TEXT_SIZE;
   char *buf = malloc(size*sizeof(char));
   int n=0;
   char temp;
   do{
       if(n >= size-2){
           char *t = realloc(buf, size+TEXT_SIZE);
           if(!t){
              printf("Ошибка: невозможно перевыделить память!");
             return 0;
           }
           size+=TEXT_SIZE;
           buf = t;
       }
       temp = getchar();
       buf[n]= temp;
       n++;
   }while(temp!='@');
   buf[n]= '\0';
   *text = buf;
   return size;
}
```

Функция выделяет память под текст и при необходимости увеличивает размер памяти, чтобы считать текст любой длины. Концом текста является знак `"@"`.

Для подсчета символов в исходном и обработанном тексте реализована функция `countUpperCase`.

```
int countUpperCase(char *text){
   int count = 0;
   for(int i=0; text[i]!='\0'; i++){
       if(isupper(text[i])){
           count++;
       }
   }
   return count;
}
```

Для удаления символов табуляции в начале предложения, а также слов верхнего регистра реализована функция `processText`. Данная функция игнорирует символы табуляции в начале каждого предложения и выделяет указатели на начало и конец предложения, которые копируются в новый массив. Аналогичным образом удаляются слова верхнего регистра.

```
char *processText(char *text, int size){
   char *new_text = malloc(size*sizeof(char));
   int src_index = 0;
   int dest_index = 0;
   int is_sentence_start = 0;
   int skip_word = 0;
   int need_copy = 0;
   for(int i=0; i<size && text[i]!='@'; i++){
  /* Пропускаем символы табуляций в начале предложения, пока
     не найдем начало предложения */
      if(text[i] == ' ' || text[i] == '\n' || text[i] == '\t'){
          continue;
       }
       else{
           if(!is_sentence_start){
               src_index = i;
               is_sentence_start = 1;
           }
       }
       /* Обрабатываем каждое слово отдельно */
       if(isalpha(text[i])){
           skip_word = processWord(text+i);
           if(skip_word > 0){
               i += skip_word;
           }
           else{
               need_copy = 1;
               i--;
           }
```

```
            }
            if(text[i] == '.')
                need_copy = 1;
            if(need_copy){
                // Вычисляем размер части для копирования
                int sentence_len = i - src_index + 1;
                if(sentence_len == 1)
    /* Если последнее слово предложения было пропущено и копируется
только точка, то она копируется на место последнего пробела */
                    dest_index--;
                if(size < dest_index + sentence_len + 1){
                    char *t = realloc(new_text, size+TEXT_SIZE);
                    if(!t){
                        printf("Ошибка: невозможно повторно
                                выделить память!");
                        return NULL;
                    }
                    size+=TEXT_SIZE;
                    new_text = t;
                }
                strncpy(new_text + dest_index, text + src_index,
sentence_len);
                /* В случае, если копируется последняя часть пред-
ложения, то необходимо добавить символ переноса строки*/
                if(skip_word >= 0){
                    new_text[dest_index + sentence_len] = '\n';
                    dest_index += 1;
                }
                dest_index += sentence_len;
                is_sentence_start = 0;
                need_copy = 0;
            }
            /* Пропускаем слово в верхнем регистре */
            if(skip_word < 0)
                i -= skip_word;
            skip_word = 0;
        }
        return new_text;
    }
```

Для определения, нужно ли вырезать слово из предложения, реализована функция `processWord`.

```
    /* Проверяем слово на содержание заглавных букв */
    int processWord(char *str){
        int i, all_upper = 1;
```

```
        for(i=0; isalpha(str[i]); i++){
            if(!isupper(str[i])){
                all_upper = 0;
            }
        }
        /* Если слово состоит только из заглавных букв,
        то возвращаем отрицательный индекс, иначе положительный */
        if(all_upper){
            return -i;
        }
        return i;
    }
```

Таким образом, если слово состоит из букв верхнего регистра, то копируются все предыдущие символы предложения, которые еще не были перенесены, а если слово содержит букву нижнего регистра, то оно просто пропускается.

В основной функции программы выводится обработанный текст, а также строка с информацией о количестве символов верхнего регистра до обработки и после:

```
    int main(){
        char *text;
        int len = readText(&text);
        int before = countUpperCase(text);
        char *processed_text = processText(text, len);
        printf("%s\n\n", processed_text);
        int after = countUpperCase(processed_text);
        printf("Количество букв в верхнем регистре до: %d и коли-
чество букв после: %d\n", before, after);
        free(text);
        free(processed_text);
        return 0;
    }
```

3.3.2. Пример выполнения задания на защиту

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

Программа на языке Си:

```
    #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 = 0;
       for(i=0; i<sizeof(vowels); i++) // проверяем с каждой глас-
ной 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 = 0; // индекс текущей ячейки для исходной строки
       int j = 0; // индекс текущей ячейки для строки-результата
       fgets(s_in, 100, stdin); // аргументы: буфер, размер бу-
фера, 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;
    }
```

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.
ОБЗОР СТАНДАРТНОЙ БИБЛИОТЕКИ

4.1. Цель и задачи

Цель работы: освоение работы со стандартной библиотекой Си.

Для достижения поставленной цели требуется решить следующие задачи:

1. Ознакомиться с заголовочными файлами стандартной библиотеки.

2. Ознакомиться с функциями стандартной библиотеки из `time.h`, `assert.h`, `ctype.h`, `string.h`, `stdlib.h`.

3. Изучить алгоритмы быстрой сортировки и бинарного поиска, а также их реализации в стандартной библиотеке.

4. Написать программу, реализующую сортировку и/или поиск по массиву в соответствии с вариантом.

5. Сформулировать выводы относительно изученных алгоритмов и роли стандартной библиотеки в языке Си.

4.2. Основные теоретические сведения

4.2.1. Структуры в языке Си

Зачастую программы оперируют огромным количеством переменных, которые как-то обрабатываются, хранятся, передаются в функции. Эти данные могут быть связаны между собой по какому-то признаку. Например, в задаче, где необходимо хранить и работать с данными о студентах, есть переменные типа `int`: количество полных лет и номер группы, а также переменные строкового типа `char *` – ФИО студента (будем считать, что ФИО не более 50 символов). Все эти переменные могут относиться к одному конкретному студенту, поэтому для хранения таких данных для 100 студентов одним из решений является создание трех массивов этих переменных:

```
int age[100];
int group[100];
char full_name[100][50];
```

Однако такой подход требует, чтобы `i`-й элемент каждого из массивов соответствовал одному и тому же студенту, что крайне неудобно и порождает ошибки.

Чтобы сгруппировать данные в языках программирования существуют более сложные типы данных, чем те, что предоставляет язык. В языке Си для этого существуют **структуры**. Пример объявления структуры:

```
struct Student{
   char full_name[50];
   int age;
   int group;
};
```

Тогда для хранения массива студентов достаточно лишь объявить массив структур:

```
struct Student students[100];
```

Проинициализировать поля структуры можно при объявлении переменной или массива.

Пример частичной инициализации полей:

```
struct Student student = {
        .age = 20,
        .full_name = "Ivan Ivanov"
};
```

Пример полной инициализации, где значения полей присваиваются по порядку их объявления в структуре:

```
struct Student student = { "Ivan Ivanov", 20, 1000};
```

Пример инициализации массива структур:

```
struct Student students[2] = {
        { "Ivan Ivanov", 20, 1000},
        {"Petr Petrov", 10, 1001}
};
```

Тогда, чтобы получить поле `full_name` i-го элемента массива используется оператор `"."`:

```
students[i].full_name;
```

Но существует оператор `"->"`, который предназначен для обращения к полям, если имеется указатель на структуру:

```
struct Student *student_ptr = &student;
student_ptr->age = 12;
```

Этот оператор позволяет не разыменовывать указатель на структуру, а сразу получать доступ к полям, т. е. заменяет запись:

```
(*student_ptr).age = 12;
```

Таким образом, структуры в языке Си позволяют группировать данные, уменьшать количество массивов и переменных и создавать произвольные типы данных.

4.2.2. Указатели на функции

Предположим, стоит задача написать универсальную функцию нахождения минимума в массиве элементов. Функция должна возвращать указатель на минимальный элемент. При этом элементами массива могут быть как числа, так и строки или структуры.

Уже известно, что указатели типа `void*` пригодятся при объявлении такой функции:

```
void* min(void*arr, int array_length, int size_of_element),
```

потому что позволяют не задумываться о конкретном типе элементов массива. Длина массива и размер элемента нужны для перебора всех элементов массива.

Попробуем реализовать алгоритм нахождения минимума:

```
void* min = arr;
for(int i=0; i < array_length; i++){
   if((arr + i*size_of_element) > min)
       min = arr + i*size_of_element;
}
```

Однако что будет результатом такого сравнения? Если `arr` – массив, то результат сравнения

```
(arr + i*size_of_element) > min
```

всегда будет 1, поскольку сравниваются указатели, а не значения. Разыменовывать указатель типа `void*` нельзя. Гипотетически можно выяснить все типы, с которыми будем работать, привести (`void*`) к каждому типу и узнавать результат сравнения отдельно для каждого типа. Но тогда функция становится неудобной в использовании и плохо масштабируемой.

Для решения подобных задач существуют **указатели на функцию**. Указатели на функцию позволяют в ходе работы программы менять вызываемую функцию. В этом примере в зависимости от типа данных можно использовать указатель на функцию-компаратор для варьирования принципа сравнения, что и рассмотрим далее.

Указатель на функцию позволяет работать с функцией как с обычной переменной, в том числе передавать функцию в качестве аргумента другой функции. Подробнее в [1], разд. 5.11.

Для написания функции нахождения минимума в массиве элементов неизвестного типа указатель на функцию нужен, чтобы сравнивать элементы. Похожая логика используется во многих языках программирования, а функцию сравнения двух элементов обычно называют **компаратором** (англ. compare – сравнивать). Компаратор работает по принципу: если элементы равны, результатом сравнения будет 0, если первый больше – результат 1 или любое положительное число, иначе –1 или любое отрицательное.

Перепишем универсальную функцию нахождения минимума с учетом указателя на функцию-компаратор:

```
void* min(void*arr, int array_length, int size_of_element,
int (*compar)(const void*,const void*)){
   void* min = arr;
```

```
    for(int i = 0; i < array_length; i++){
        if(compar(arr + i*size_of_element, min) < 0)
            min = arr + i*size_of_element;
    }
    return min;
}
```

Теперь можно вызывать функцию `min` для любого типа, главное – определить функцию сравнения двух элементов.

Реализуем компаратор, который сравнивает два целых числа:

```
int cmp(const void *a, const void *b){
    const int *first = (const int *)a;
    const int *sec = (const int *)b;
    if(*first > *sec)
        return 1;
    if(*first < *sec)
        return -1;
    return 0;
}
```

Пример поиска минимального значения в массиве:

```
int main(void){
    int arr[] = {1, -3, 2, -5, 10, 2};
    int *res = min(arr, sizeof(arr)/sizeof(int), sizeof(int), cmp);
    printf("%d\n", *res);
    return 0;
}
```

Результат программы – число –5, которое является минимальным в массиве.

4.2.3. Обзор стандартной библиотеки

Рассмотрим функции и заголовочные файлы стандартной библиотеки Си `libc`, которые необходимы для выполнения лабораторной работы. Рассмотрение ряда других заголовочных файлов стандартной библиотеки приведено в приложении.

Обработка строк и символов. Стандартная библиотека Си предоставляет большие возможности для работы со строками и символами, которые определены в следующих заголовочных файлах: `string.h`, `ctype.h`, `wchar.h`, `wctype.h`, `locale.h`, `uchar.h`.

Широкие символы. Стандартный тип `char`, используемый для хранения символов имеет размер 1 байт и позволяет задать любой символ из таблицы ASCII. И если за латинскими символами, управляющими (`\n`, `\0` и т. п.) и некото-

рыми другими (цифры, скобки) закреплены свои коды в первой части таблицы, то для символов национальных алфавитов и других, менее распространенных символов, одной таблицы ASCII недостаточно. Решение этой проблемы – использование других таблиц кодировок, на код одного символа в которых отводится больше 1 байта, но для этого использование типа `char` уже не подходит.

Для работы с широкими символами в программах на языке Си используется тип `wchar_t`. Важно понимать, что фактическое представление символов в `wchar_t` не регламентировано и может отличаться в зависимости от платформы.

В ОС Линукс `wchar_t` обычно имеет размер 4 байта и хранит символы в соответствии с UTF-32. Для использования типа `wchar_t`, требуется подключить заголовочный файл `wchar.h`.

Работа с типом `wchar_t` несколько похожа на работу с уже знакомым ранее типом `char`, но если при объявлении строки типа `wchar_t` отличий нет:

```
wchar_t s1[100];
```

то при использовании строковых литералов, которые должны быть представлены в `wchar_t`, требуется перед такой строкой или символом добавить букву `L`:

```
wchar_t s2[] = L"wchar_t hello!";
if(s2[0] == L'w')
{
   ...
}
```

Для работы с `wchar_t` существует специальный набор функций, аналогичный тому, который используется для строк и символов `char`, включая функции ввода/вывода. Функции для строк содержатся в заголовочном файле `wchar.h`, а для символов в файле `wctype.h`.

(!) Важное замечание. Для считывания кириллических символов с клавиатуры и вывода их на экран при использовании `wchar_t`, следует установить локаль по умолчанию с помощью вызова функции `setlocale` (подробнее о локалях [20]):

```
setlocale(LC_CTYPE, "");
```

Для этого следует подключить заголовочный файл `locale.h`. Это необходимо для корректного отображения кириллических символов. Локаль определяет кодировку символов, т. е. определяет соответствие числового значения символа с буквой алфавита того или иного языка. Для каждого языка в различных странах могут использоваться различные локали.

Пример программы, считывающей с клавиатуры строку, содержащую кириллицу, переводящую ее в верхний регистр и дописывающую ее к строке `Пользователь ввел`:

```
#include <stdio.h>
#include <wchar.h>
#include <wctype.h>
#include <locale.h>

int main()
{
   int i;
   wchar_t s[150] = L"Пользователь ввел: ";
   wchar_t s1[100];
   setlocale(LC_CTYPE, "");
   fgetws(s1,sizeof(s1)/sizeof(wchar_t),stdin);
   for(i=0;i<wcslen(s1);i++)
       s1[i] = towupper(s1[i]);
   wcscat(s,s1);
   wprintf(L"%ls",s);
   return 0;
}
```

Работа с символами. В состав стандартной библиотеки входят функции для работы с символами, объявленные в заголовочном файле `ctype.h` и `wctype.h` для широких символов соответственно.

Эти функции позволяют переводить символы из верхнего регистра в нижний и определять, является ли этот символ буквой, цифрой или каким-то служебным.

Пример: если требуется сравнить два символа `ch1` и `ch2` без учета регистра, можно выполнить это следующим образом:

```
if(toupper(ch1)==toupper(ch2)){
   ...
}
```

Работа со строками и памятью. Стандартная библиотека Си содержит различные функции для работы с памятью и строками, объявленные в заголовочном файле `string.h`

Среди них есть функции:

- копирования памяти (`memcpy`, `memmove`);
- сравнения памяти (`memcmp`);
- сравнения строк (`strcmp`, `strncmp`);
- разбиения строки на токены (`strtok`);

• конкатенации строк (strncat);

• поиска символов и подстроки в строке (strchr, strstr);

• определения длины строки (strlen) и некоторые другие.

Также для преобразования из короткого символа в широкие существуют специальные функции в uchar.h.

Обработка различных типов данных и работа с памятью. В заголовочном stdlib.h файле собраны объявления различных функций, часть из которых уже использовалась ранее:

• функции для работы с динамической памятью;

• функции для преобразования строки в число;

• генерации псевдослучайных чисел;

• функции для управления процессом выполнения программы;

• функции для вычисления абсолютного значения и деления целых чисел;

• функции для сортировки и поиска.

На последние две функции стоит обратить особое внимание: эти функции позволяют выполнять быструю сортировку (qsort) и бинарный поиск (bsearch) в массиве данных любого типа.

Алгоритм быстрой сортировки и бинарного поиска (qsort и bsearch). Как уже говорилось, в stdlib.h есть функции для сортировки и поиска в массиве любого типа. Рассмотрим как такое возможно на примере функции qsort:

```
void qsort (void* base, size_t num, size_t size,
            int (*compar)(const void*,const void*));
```

Функция принимает указатель на начальный элемент массива, количество элементов и размер одного элемента, а также указатель на функцию для сравнения двух элементов.

Так как тип элементов может быть любым, то и указатель на первый элемент массива имеет тип void. Это позволяет, зная адрес первого элемента и размер каждого элемента, вычислить адрес любого элемента массива в памяти с помощью арифметики указателей и обратиться к нему. Остается только сравнить два элемента, имея два указателя на них. Это выполняет функция compar, указатель на которую передается функции qsort в качестве одного из параметров.

Функция compar принимает два указателя типа void, но в своей реализации может привести их к конкретному типу (так как её реализация остается за программистом, он точно знает элементы какого типа он сортирует) и сравнить их. Результат сравнения определяется знаком возвращаемого функций qsort числа.

Рассмотрим простейший пример сортировки массива чисел:

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

int cmp(const void *a, const void *b){
   const int *first = (const int *)a;
   const int *sec = (const int *)b;
   if(*first > *sec)
       return 1;
   if(*first < *sec)
       return -1;
   return 0;
}

int main(){
   int arr[] = {1, 4, -4, 6, 2, 0};
   int n = sizeof(arr)/sizeof(int);
   qsort(arr, n, sizeof(int), cmp);
   for(int i=0; i<n; i++)
       printf("%d ", arr[i]);
   printf("\n");
   return 0;
}
```

Этот пример сортирует массив по возрастанию, а чтобы отсортировать массив по убыванию, необходимо лишь поменять знаки в условиях компаратора:

```
int cmp(const void *a, const void *b){
   const int *first = (const int *)a;
   const int *sec = (const int *)b;
   if(*first < *sec)
       return 1;
   if(*first > *sec)
       return -1;
   return 0;
}
```

Так как в динамическом массиве может храниться любая структура данных, то встает вопрос: как сортировать структуры в языке Си? Разница в сортировке массива структур от массива стандартных типов данных лишь в компараторе, который определенным разработчиком образом будет сравнивать структуры. Рассмотрим пример структуры для студента:

```
struct Student{
   char name[50];
   char lastname[50];
   int age;
};
```

Чтобы отсортировать массив из таких структур:

- лексикографически по фамилии;
- лексикографически по имени;
- по возрасту,

необходимо сравнивать поле фамилии, затем сравнивать поле имени в случае, если фамилии совпадают, и в случае совпадения имени и фамилии сравнить возраст студентов. Подобный компаратор может выглядеть следующим образом:

```
int cmp(const void *a, const void *b){
   const struct Student *first = (struct Student *)a;
   const struct Student *sec = (struct Student *)b;
   int res = strcmp(first->lastname, second->lastname);
   if(res) return res;
   res = strcmp(first->name, second->name);
   if(res) return res;
   if(first->age < second->age) return -1;
   else if(first->age > second->age) return 1;
   return 0;
}
```

Тогда вызов функции `qsort` будет выглядеть:

```
qsort(arr, n, sizeof(struct Student *), cmp);
```

Функция для поиска элементов `bsearch` очень похожа по своей сигнатуре на `qsort`:

```
void bsearch(void *key, void* base, size_t num, size_t size,
             int (*compar)(const void*,const void*));
```

Эта функция ищет в отсортированном массиве `base` элемент `key` и также принимает на вход функцию-компаратор. Функция `bsearch` возвращает NULL, если элемент не найден, или возвращает адрес элемента в противном случае. В отличие от `qsort` компаратор в `bsearch` на вход в качестве первого элемента всегда получает `key`, а в качестве второго – элемент массива.

Таким образом, предыдущую программу с сортировкой массива можно дополнить следующим кодом для поиска элемента:

```
int key = -4;
int *result = (int *)bsearch(&key, arr, n, sizeof(int), cmp);
```

Аналогичным образом можно сортировать многомерные массивы и динамические массивы строк. Приведем пример сортировки и поиска элементов в динамическом массиве строк:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int cmp(const void *a, const void *b){
   const char **f = (const char **)a;
   const char **s = (const char **)b;
   return strcmp(*f, *s);
}

int search_cmp(const void *key, const void *val){
   const char *k = (const char *)key;
   const char **v = (const char **)val;
   return strcmp(k, *v);
}

int main(){
   srand(time(NULL));
   char **strs = malloc(5*sizeof(char *));
   for(int i=0; i<5; i++){
       strs[i] = calloc(50, 1);
       memset(strs[i], 'A' + rand()%25, 10);
   }
   qsort(strs, 5, sizeof(char *), cmp);
   for(int i=0; i<5; i++){
       printf("%s\n", strs[i]);
   }
   char *key = "EEEEEEEEEE";
   char **res = (char **)bsearch(key, strs, 5, sizeof(char *),
                 search_cmp);
   if(res) printf("%ld\n", res-strs);
   else printf("Not found\n");
   return 0;
}
```

Программа генерирует пять случайных строк с латинскими символами, сортирует массив строк и ищет в этом массиве строку `"EEEEEEEEEE"`. Выводит индекс этой строки, если она найдена. Как можно заметить, необходимо писать разные компараторы для поиска и сортировки элементов, а также стоит помнить, что порядок аргументов компаратора для функции `bsearch` важен, поскольку ключ подается первым аргументом, а вторым – значение из массива для сравнения.

Расширение стандартных типов данных. В языке Си не предусмотрен тип данных `bool`, и явно не указана разрядность целых чисел, поэтому в состав

библиотеки входят заголовочные файлы: `stdbool.h` – определение типа `bool` и значений `true`, `false`, `stdint.h` – определения знаковых и беззнаковых целочисленных типов данных определенной разрядности. `stdint.h` позволяет фиксировать размер целого числа в битах на любой платформе, если платформа поддерживает вычисления значений такой разрядности.

В `inttypes.h`, `float.h`, `fenv.h` объявлены макросы для работы с типом `intmax_t` и `float`. Для получения информации о нижней и верхней границах значений для различных типов данных существует заголовочный файл `limits.h`

Работа с датой и временем. В заголовочном файле `time.h` можно найти объявления типов и функций для работы с датой и временем. В том числе:

- функцию, позволяющая получить текущее календарное время (`time`);
- функцию, позволяющая получить время в тактах процессора с начала выполнения программы (`clock`);
- функцию преобразования времени в стандартную форму UTC (`gmtime`);
- функцию преобразования времени в местное время (`localtime`);
- функцию для вычисления разности в секундах между двумя временными штампами (`difftime`);
- функцию для вывода значения даты и времени на экран (`asctime`, `ctime`);

А также структуру `tm`, содержащих компоненты календарного времени и функцию для преобразования значения времени в секундах в объект такого типа.

Подробнее о стандартной библиотеке в [21].

4.3. Задание к лабораторной работе 4

В этой работе необходимо разработать программу, которая реализует сортировку и/или поиск в массиве или строке (массиве символов) в соответствии с вариантом. В каждом варианте представлена задача, для решения которой необходимо использовать функции стандартной библиотеки.

4.3.1. Описание последовательности выполнения работы

В работе рассматриваются два основных типа заданий:

1) сортировка и поиск в массиве целых чисел

2) обработка строк, сортировка и поиск в динамическом массиве строк

В качестве примера рассмотрим следующий вариант задания: напишите программу, которой на вход подается строка, состоящая из произвольного набора символов.

Программа должна реализовать следующий алгоритм:

1) считать строку из стандартного потока ввода `stdin`;

2) отсортировать строку по возрастанию значений кодов ASCII с помощью алгоритма "быстрая сортировка" (`quick sort`), используя при этом функцию стандартной библиотеки;

3) посчитать время, за которое будет отсортирован массив строк, используя при этом функцию стандартной библиотеки;

4) вывести отсортированный массив (элементы массива должны быть разделены пробелом);

5) вывести время, за которое был отсортирован массив строк.

Каждый символ соответствует коду в таблице ASCII, и по сути этот код и хранится в качестве значения `char`. Поэтому для написания компаратора достаточно просто сравнивать числа, хранящиеся в значениях двух символов. Числа можно сравнивать, вычитая из первого числа второе. В таком случае компаратор будет выглядеть как

```
int compar(const char *a, const char *b){
   return *a - *b;
}
```

Стоит отметить, что этот компаратор работает корректно в случае `char`-значений, но если написать аналогичный компаратор для `int`-значений, то могут возникнуть ситуации, когда компаратор будет возвращать некорректное значение из-за переполнения. Операции с `char` производятся после перевода значений в `int`, поэтому переполнения возникнуть не может.

Пример корректного компаратора для целых значений приведен в 4.2.3 "Алгоритм быстрой сортировки и бинарного поиска (`qsort` и `bsearch`) ".

Далее была написана функция `main`, которая считывает строку с помощью `fgets` и сортирует данные с помощью `qsort`. В случае ошибки при чтении строки программа возвращает ошибку ввода-вывод `EIO`.

```
int main(){
   char str[STR_LEN];
   char *str_ptr = fgets(str, STR_LEN, stdin);
   if(str_ptr == NULL){
       printf("Input error!\n");
       return 0;
   }
   unsigned int time_start=clock();
   qsort(str, strlen(str), sizeof(char), (int (*) (const void *,
const void *)) compar);
   unsigned int sort_time=clock() - time_start;
```

```
    printf("%s\n", str);
    printf("\n%f\n", ((float)sort_time)/CLOCKS_PER_SEC);
    return 0;
}
```

Поскольку константы в языке Си принято заменять макроподстановками, то длина массива символа была объявлена как

```
#define STR_LEN 1000
```

Вывод: в результате разработана программа для сортировки символов в строке по их ASCII-кодам. Для достижения поставленной цели был изучен состав стандартной библиотеки Си, и на практике использованы функции ввода-вывода, сортировки и работы со временем.

4.3.2. Пример выполнения задания на защиту

Задание. Написать программу, которая выведет на экран дату, наступающую через 1 год, 33 дня, 12 часов и 53 минуты. Формат вывода должен быть следующим: "Sat Jul 16 2:03:55 1994\n\0" (для данного форматного вывода необходимо использовать специальную функцию стандартной библиотеки). Использовать часовой пояс по умолчанию в системе.

Код программы:

```
int main(void)
{
    time_t current_time;
    struct tm *date;

    current_time = time(NULL);
    date = localtime(&current_time);
    date -> tm_year += 1;
    date -> tm_mday += 33;
    date -> tm_hour += 12;
    date -> tm_min += 53;

    time_t res_time = mktime(date);

    printf("%s", asctime(date));
}
```

Вопросы для контроля

1. В каком заголовочном файле стандартной библиотеки расположены функции для работы с символами?

2. Зачем нужны широкие символы и чем они отличаются от стандартных символов `char` языка Си?

3. Какие должны быть ограничения на входные данные, чтобы бинарный поиск работал корректно и мог быть использован?

5. Курсовая работа.
ОБРАБОТКА ТЕКСТОВЫХ ДАННЫХ

5.1. Цель и задачи

Цель курсовой работы – освоить подходы к работе с текстовыми данными в языке Си. Достижение этой цели включает следующие задачи:

1. Разработать способ представления и хранения текста, предложений и слов в памяти.

2. Реализовать алгоритм считывания текста произвольной длины с клавиатуры и вывода его на экран.

3. Освоить способ выделения из текста отдельных его частей: подстрок, являющихся словами, частями слов, предложениями.

4. Освоить способ модификации символов текста или его частей. Удаление, добавление новых символов.

5. Выполнить задания курсовой работы на обработку текста в соответствии с условием варианта.

5.2. Основные теоретические сведения

Зачастую при считывания из потока данных `stdin` неизвестна длина входных данных, поэтому возникают задачи считывания данных неизвестного размера двух видов:

1. Строго структурированные данные, которые имеют известные типы и разделители при вводе.

2. Неструктурированные данные, которые могут быть разделены любыми (или ограниченным количеством) символами (пробелы, запятые, точки и т. д.).

Рассмотрим случай, когда на вход подается текст с предложениями разделенными точкой, восклицательным знаком или символом переноса строки `"\n"`. Для этого реализуется функция чтения предложения из стандартного потока ввода и структуру, которая будет хранить в себе предложение. При считывании предложения не известна длина этого предложения, поэтому считывание происходит с некоторым шагом. Пусть, например, шаг будет равен пяти символам, т. е. в предложение будет добавляться по пять символов, перед расширением памяти с помощью функции `realloc`. Структура `Sentence` будет хранить в себе строку, куда запишется предложение и размер массива символов (нашей строки), который не равен длине строки, а равен количеству байт, выделенных под эту строку.

```
#define MEM_STEP 5
struct Sentence{
   char *str;
   int size; // != strlen(str)
};
```

Функция `readSentence` будет возвращать указатель на предложение. В первую очередь выделяется память под строку, куда записывается предложение в соответствии с выбранным шагом. На каждом считанном символе необходимо проверить размер буфера строки с количеством считанных символов, чтобы перевыделить память, если ее не хватает. Точкой останова является считанный и записанный в буфер символ переноса строки, точка и восклицательный знак. Указанный размер и указатель на строку с предложением копируются в соответствующие поля структуры созданной `Sentence`.

```
struct Sentence* readSentence(){
   int size = MEM_STEP;
   char *buf = malloc(size*sizeof(char));
   int n=0;
   char temp;
   do{
       if(n >= size-2){
           char *t = realloc(buf, size+MEM_STEP);
           if(!t){ /* ОШИБКА */}
           size+=MEM_STEP;
           buf = t;
       }
       temp = getchar();
       buf[n]= temp;
       n++;
   }while(temp!='\n' && temp!='.' && temp!='!');
   buf[n]= '\0';
   struct Sentence *sentence = malloc(sizeof(struct Sentence));
   sentence->str = buf;   //(*sentence).str = buf;
   sentence->size = size;
   return sentence;
}
```

Умея читать предложение неизвестной длины можно читать текст с неизвестным числом предложений. Точкой останова является два подряд идущих символа переноса строки. Структура `Text` содержит двумерный массив предложений, количество предложений в тексте и размер буфера под двумерный массив.

```
struct Text{
   struct Sentence **sents;
   int n;
   int size;
};
```

Аналогично чтению предложения в начале выделяется память под массив предложений и происходит пошаговое считывание предложений с помощью функции `readSentence`. В данной реализации пропускаются строки с символом `"\n"`.

```
struct Text readText(){
   int size = MEM_STEP;
   struct  Sentence**  text=  malloc(size*sizeof(struct  Sen-
tence*));
   int n=0;
   struct Sentence* temp;
   int nlcount = 0;
   do{
       temp = readSentence();
       if(temp->str[0] == '\n')
           nlcount++;
       free(temp);
       else{
           nlcount = 0;
           text[n] = temp;
           n++;
       }
   }while(nlcount<2);
   struct Text txt;
   txt.size = size;
   txt.sents = text;
   txt.n = n;
   return txt;
}
```

Подробнее про структуры можно почитать в книге одного из создателей языка программирования C++ Страуструпа "Программирование. Принципы и практика использования C++" (см. [22]).

5.3. Задание к курсовой работе

Считать текст произвольной длины. В этом тексте найти все предложения, размер которых больше 20 символов. Концом текста являются два символа переноса строки `"\n"`. Любые символы табуляции, переноса строки и пробелы в

начале предложения не являются частью предложения. Необходимо реализовать выбор действия из консоли в соответствии со следующими двумя пунктами:

1. Вывод количества предложений, где количество символов превышает 20 единиц.

2. Вывод предложений, где количество символов превышает 20 единиц.

5.3.1. Описание последовательности выполнения работы

Для выполнения курсовой работы необходимо реализовать программу, которая считывает текст и позволяет выбрать опцию для вывода результата в соответствии с заданием.

В начале необходимо определить, как будет представлен входной текст в программе и установить шаг, на который будет увеличиваться размер буфера при чтении данных. Предложение удобно представлять в виде структуры, состоящей из полей `str` и `size`, которые позволяют хранить строку и размер буфера под эту строку. Текст в таком случае – двумерный массив, но его также удобно представить в виде структуры из двумерного массива предложений, размера этого массива и количества предложений, которые там хранятся:

```
#define MEM_STEP 5

struct Sentence{
   char *str;
   int size;
};

struct Text{
   struct Sentence **sents;
   int n;
   int size;
};
```

Для чтения предложения реализована функция `readSentence`, которая игнорирует пробелы, табуляции и символы переноса строки в начале предложения, затем считывает само предложение до точки, восклицательного знака или символа переноса строки. В случае ошибки выводится соответствующая строка:

```
struct Sentence* readSentence(){
   int size = MEM_STEP;
   char *buf = malloc(size*sizeof(char));
   int is_sent_start = 1;
   int n=0;
   char temp;
   do{
```

```
        if(n >= size-2){
            char *t = realloc(buf, size+MEM_STEP);
            if(!t){
                printf("Ошибка, ну удается выделить память!\n");
                return NULL;
            }
            size+=MEM_STEP;
            buf = t;
        }
        temp = getchar();
        if(is_sent_start){
            if(temp == '\t' || temp == ' '){
                continue;
            }
            else{
                is_sent_start = 0;
            }
        }
        buf[n]= temp;
        n++;
    }while(temp!='\n' && temp!='.' && temp!='!');
    buf[n]= '\0';
    struct Sentence *sentence = malloc(sizeof(struct Sentence));
    sentence->str = buf;
    sentence->size = size;
    return sentence;
}
```

Для чтения текста реализована функция `readText`, которая последовательно считывает предложения, пока не встретит два подряд идущих символа переноса строки:

```
struct Text readText(){
    int size = MEM_STEP;
    struct Sentence** text= malloc(size*sizeof(struct Sentence*));
    int n=0;
    struct Sentence* temp;
    int nlcount = 0;
    do{
        temp = readSentence();
        if(temp->str[0] == '\n'){
            nlcount++;
            free(temp);
        }
```

```
        else{
            nlcount = 0;
            text[n] = temp;
            n++;
        }
    }while(nlcount<2);
    struct Text txt;
    txt.size = size;
    txt.sents = text;
    txt.n = n;
    return txt;
}
```

Для подсчета строк длиной более 20 символов реализована функция `countAllStringMoreThan20`, а для вывода этих строк – `printMoreThan20`:

```
void printMoreThan20(struct Text t){
    for(int i=0; i<t.n; i++){
        struct Sentence *cur = t.sents[i];
        if(strlen(cur->str)>20)
            printf("%s\n", cur->str);
    }
}
int countAllStringMoreThan20(struct Text t){
    int count = 0;
    for(int i=0; i<t.n; i++){
        struct Sentence *cur=t.sents[i];
        if(strlen(cur->str)>20)
            count++;
    }
    return count;
}
```

В конце программы необходимо очистить всю выделенную память, чтобы избежать утечек памяти. Для этого в функции `freeText` происходит обход всех предложений с освобождением памяти под структуру предложения и под строку для этого предложения. В последнюю очередь освобождается память двумерного массива предложений:

```
void freeText(struct Text t){
    for(int i=0; i<t.n; i++){
        free(t.sents[i]->str);
        free(t.sents[i]);
    }
    free(t.sents);
}
```

В главной функции программы происходит считывание текста с помощью уже описанных функций, а затем выбор функции вывода в соответствии с выбором пользователя:

```
int main(){
    printf("Введите текст:\n");
    struct Text t = readText();
    printf("Введите цифру 1 для вывода информации о количестве предложений длины более 20 символов, цифру 2 для вывода этих предложений.\n");
    int action;
    scanf("%d", &action);
    switch(action){
        case 1:
            printf("Количество предложений длины более 20 символов: %d\n", countAllStringMoreThan20(t));
            break;
        case 2:
            printf("Предложения длины более 20 символов:\n");
            printMoreThan20(t);
            break;
    }
    freeText(t);
    return 0;
}
```

На защите курсовой работы нужно выполнить задание, аналогичное пунктам из курсовой работы. Например, необходимо расширить выбор действий в курсовой работе и добавить вывод средней длины предложений в тексте.

Список использованной литературы

1. Керниган Б., Ритчи Д. Язык программирования Си / пер. с англ, 3-е изд., испр. СПб.: Невский Диалект, 2004.
2. gcc(1) – Linux manual page. URL: https://man7.org/linux/man-pages/man1/gcc.1.html
3. Gough B. J., Stallman R. An Introduction to GCC. Network: Theory Limited, 2004.
4. Шилдт Г. C. 2.0. Полное руководство / пер. с англ. М.: Вильямс, 2011.
5. Прата С. Язык программирования С: лекции и упражнения. Litres, 2022.
6. МакГрат М. Программирование на С для начинающих. Litres, 2022.
7. GDB: The GNU Project Debugger. URL: https://www.sourceware.org/gdb/
8. Клеменс Б. Язык С в XXI веке. Litres, 2022.
9. Фомин С., Подбельский В. Курс программирования на языке Си: учебник. Litres, 2022.
10. Burgess M., Hale-Evans R. The GNU C Programming Tutorial. 2002.
11. Онлайн справочник программиста на C и C++. URL: https://www.cplusplus.com/reference/cstdio/printf/
12. The C Preprocessor : Standard Predefined Macros . URL: https://gcc.gnu.org/onlinedocs/cpp/Standard-Predefined-Macros.html
13. Logging with GCC (материал повышенной сложности). URL: https://www.valvers.com/open-software/logging-with-gcc/
14. The entire table of ANSI color codes. URL: https://gist.github.com/iamnewton/8754917
15. C Programming / Linux – Color text output. URL: https://ramprasadk.wordpress.com/2010/06/09/c-programming-linux-color-text-output/
16. Васильев А. Н. Программирование на С в примерах и задачах. Эксмо-Пресс, 2020.
17. Hall B. Beej’s Guide to C Programming. 2021.
18. Oualline S. C Elements of Style. M&T Books, 1992.

Список рекомендуемой литературы

Дейтел П., Дейтел Х. С для программистов с введением в С11. Litres, 2022.

Документация для Linux. Локализация и Интернационализация. URL: https://www.opennet.ru/docs/RUS/cyr_howto/ch06.html

Страуструп Б. Программирование. Принципы и практика использования C++. Litres, 2022.

Edsger Wybe Dijkstra, "Go To Statement Considered Harmful", Communications of the ACM. Vol. 11, № 3, March 1968. P. 147–148.

Rothwell T., Youngman J. The gnu c reference manual // Free Software Foundation, Inc. 2007. С. 86.

ПРИЛОЖЕНИЕ

ОБЗОР СТАНДАРТНОЙ БИБЛИОТЕКИ

1. Математические функции

Для различных математических вычислений в библиотеку `libc` входят: `math.h`, `tgmath.h`, `complex.h`. Функции взятия корня, округления, тригонометрические функции и т. д. реализованы в `math.h`, а работа с комплексными числами – в `complex.h`.

Так как в `math.h` функции зависят от входных параметров и предназначены для определенных типов данных, например, `abs` – получение абсолютного значения для целых чисел и `fabs` – для чисел с плавающей точкой, то в стандартную библиотеку были добавлены макросы, которые позволяют определять типы данных и подставлять нужную функцию. Такие макросы реализованы в `tgmath.h`

2. Отладка и обработка ошибок

В стандартной библиотеке объявлен макрос препроцессора `assert()` в `assert.h`. С его помощью можно выполнять проверку некоторых условий в процессе выполнения программы.

Пример:

```
int my_inc(int a){
   return a+1;
}

int main(){
   assert(my_inc(3) == 4);
}
```

Если условие ложно, то в процессе выполнения будет выведена некоторая информация о том, в каком месте это произошло (имя файла с исходным кодом, имя функции, номер строки) и само условие.

Например, для строчки:

```
assert(my_inc(3) != 4);
```

программа выведет:

```
a.out: main.c:16: main: Assertion `my_inc(3) != 4' failed.
```

Стоит заметить, что для отключения проверок достаточно всего лишь добавить макрос `#define NDEBUG` перед включением заголовочного файла `assert.h`.

Использование `assert` может сильно упростить отладку программ, гарантируя, что все необходимые условия в процессе выполнения программы соблюдены.

Большинство функций возвращают код ошибки, но не всегда это возможно, так как функция может ничего не возвращать или возвращаемый тип данных не позволяет определить ошибку по своему значению, например функция десятичного логарифма log10, может возвращать как положительные, так и отрицательные значения, но на вход получать отрицательное значение она не может, поэтому необходимо каким-то образом сообщить о возникшей ошибке. Для этого существует специальная переменная `errno` в заголовочном файле `errno.h`. Также там объявлены функции вывода ошибок и символьные имена ошибок.

Пример обработки таких ошибок:

```
int main(void)
{
   log10(-1);
   if (errno==EDOM)
       perror ("Program Error");
   return 0;
}
```

Результатом программы будет следующий вывод в консоль:

```
Program Error: Numerical argument out of domain
```

3. Параллельное выполнение

Для реализации многопоточности или ее имитации в стандартную библиотеку входят `stdatomic.h`, `threads.h`. В рамках курса по программированию библиотеки `stdatomic.h`, `threads.h` рассматриваться не будут.

4. Прочее

Помимо ранее рассмотренных заголовочных файлов в стандартной библиотеке находятся:

1) `stdio.h` – уже известный заголовочный файл с функциями ввода-вывода;

2) `stdarg.h` содержащий средства для обработки аргументов функции;

3) `signal.h` содержащий средства для обработки сигналов;

4) `time.h` содержащий функции для работы с датой и временем;

5) `iso646.h` содержащий символьные замены побитовым и логическим операциям, которые необходимы для некоторых не qwerty-клавиатур;

6) `stdalign.h` содержащий средства для выравнивания данных;

7) `stddef.h` содержащий такие макроопределения, как NULL, `size_t` и др.;

8) `stdnoreturn.h` содержащий макрос `noreturn`, который показывает невозвратные функции – функции, которые осуществляют выход из программы;

9) `setjmp.h` содержит функции для безусловного перехода между функциями.

Далее будет рассмотрена только основная часть заголовочных файлов, которые наиболее часто используются.

Ввод-вывод

В заголовочном файле `stdio.h` содержатся уже известные нам функции чтения из потока ввода и вывода информации в стандартный поток вывода. Также в нем содержатся функции для работы с файлами и функции для указания собственного буфера указанного потока. Ранее уже были описаны различные функции чтения из стандартного потока ввода, например функция `fgets`, которая также позволяет читать из файла, если указать открытый файл `FILE *`.

Для работы с файлами выделены следующие функции:

- `fopen`, которая позволяет открыть файл для операций над ним;
- `fclose`, которая закрывает файл;
- `remove`, которая удаляет файл;
- `rename`, которая позволяет переименовать файл;
- `tmpfile`, которая создает временный файл и удаляет его при вызове `fclose`.

Стандартные потоки ввода, вывода и ошибок также представлены в виде констант типа *FILE* *: `stdin`, `stdout`, `stderr`. Стоит отметить, что для отладки рекомендуется использовать поток `stderr`, поскольку данный поток сразу выводит информацию в отличие от потока `stdout`, который буферизирует всю информацию для вывода. Если программа была завершена по какой-либо ошибке до вывода буфера, то он будет утерян в случае `stdout`.

Обработка произвольного числа аргументов функции

В языке Си возможно использование функций с произвольным числом и типом аргументов (вспомним функцию `printf`, например). Для удобства работы с такими аргументами есть средства, находящиеся в заголовочном файле `stdarg.h`.

Заголовочный файл определяет специальный тип `va_list` и набор функций для работы с этим типом `va_start`, `va_arg`, `va_end`, с помощью которых можно по очереди перебирать аргументы функции.

Обработка сигналов

Программы на вход помимо потока ввода могут также получать различные сигналы от системы и/или пользователя, например, нажатие `CTRL+C`. Чтобы задать поведение программы при получении различных сигналов, в стандартную библиотеку включен заголовочный файл `signal.h`. В нем содержатся функции для перехвата и обработки сигналов.

Безусловный переход между функциями

В языке Си существует оператор безусловного перехода `goto`, но он имеет ограничения, так как не позволяет переходить из одной функции в другую. Не будем затрагивать причины этих ограничений, но для решения задачи перехода между функциями в стандартную библиотеку был добавлен заголовочный файл `setjmp.h`, который содержит всего две функции:

- `int setjmp(jmp_buf envbuf)` – функция сохранения текущего состояния, которая является аналогом метки для `goto`. Именно к последнему вызову этой функции и будет передано управление с помощью второй функции `longjmp`;
- `void longjmp(jmp_buf, int val)` – функция передает управление в последний вызов `setjmp` и позволяет вернуть значение `val`. Значение `val` будет возвращено в повторном вызове функции `setjmp`.

Также стоит отметить, что еще Эдсгер Вайб Дейкстра в 1968 г. сформулировал, почему оператор `goto` использовать нежелательно (см. [23]). А заголовочный файл `setjmp.h` может использоваться для различных ситуаций перехода между функциями, например, имитации системы исключений, таких как try – catch в C++. Но данный заголовочный файл стоит использовать только тогда, когда это действительно необходимо, как и `goto`.

СОДЕРЖАНИЕ

Кринкин Кирилл Владимирович,
Берленко Татьяна Андреевна,
Заславский Марк Маркович,
Чайка Константин Владимирович,
Допира Валерия Евгеньевна,
Гаврилов Андрей Владимирович

Базовые сведения к выполнению курсовой и лабораторных работ по дисциплине "Программирование". Первый семестр

Учебно-методическое пособие

Редактор О. Р. Крумина

Компьютерная верстка Е. Н. Стекачевой

Подписано в печать 29.12.22. Формат 60×84 1/16.
Бумага офсетная. Печать цифровая. Печ. л. 5,25.
Гарнитура "Times New Roman". Тираж 50 экз. Заказ .

Издательство СПбГЭТУ "ЛЭТИ"
197022, С.-Петербург, ул. Проф. Попова, 5Ф