

# Очередное введение в язык программирования С

Иван Овчинников

28 января 2022 г.

# Содержание

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Введение</b>  | <b>4</b>  |
| 1.1      | От автора . . . . .  | 4         |
| 1.2      | Инструментарий . . . . .                                       | 4         |
| 1.3      | Немного истории . . . . .                                      | 5         |
| <b>2</b> | <b>Основные понятия</b>  | <b>6</b>  |
| 2.1      | Компиляция . . . . .   | 6         |
| 2.2      | Шаблон программы . . . . .                                     | 7         |
| <b>3</b> | <b>Базовый ввод-вывод</b>                                      | <b>10</b> |
| 3.1      | Форматированный вывод . . . . .                                | 10        |
| 3.2      | Форматированный ввод . . . . .                                 | 13        |
| <b>4</b> | <b>Переменные и типы данных. Базовые манипуляции с данными</b> | <b>15</b> |
| 4.1      | Переменные в программе на языке C . . . . .                    | 15        |
| 4.2      | Базовая арифметика . . . . .                                   | 19        |
| 4.3      | Булева алгебра и двоичные вычисления . . . . .                 | 22        |
| <b>5</b> | <b>Условия, блоки кода, видимость</b>                          | <b>28</b> |
| 5.1      | Условный оператор . . . . .                                    | 28        |
| 5.2      | Операции сравнения . . . . .                                   | 32        |
| 5.3      | Блоки кода и область видимости . . . . .                       | 34        |
| <b>6</b> | <b>Циклы</b>   | <b>36</b> |
| 6.1      | Операторы циклов . . . . .                                     | 36        |
| 6.1.1    | <code>while()</code> . . . . .                                 | 36        |
| 6.1.2    | <code>do {} while();</code> . . . . .                          | 38        |
| 6.1.3    | <code>for(;;)</code> . . . . .                                 | 39        |
| 6.2      | Управление циклами . . . . .                                   | 40        |
| 6.3      | Практические задачи . . . . .                                  | 41        |
| 6.4      | Множественный выбор <code>switch(){} </code> . . . . .         | 45        |
| <b>7</b> | <b>Функции</b>   | <b>49</b> |
| 7.1      | Понятие функции, параметры и аргументы . . . . .               | 49        |
| 7.2      | Оформление функций. Понятие рефакторинга . . . . .             | 52        |

|           |  |           |
|-----------|--|-----------|
| 7.3       | Прототип функции, заголовочные файлы . . . . .                     | 54        |
| <b>8</b>  | <b>Указатели</b>   | <b>56</b> |
| <b>9</b>  | <b>Массивы</b>   | <b>60</b> |
| 9.1       | Директива #define . . . . .  | 60        |
| 9.2       | Массивы . . . . .  | 61        |
| 9.3       | Идентификатор массива . . . . .                                    | 65        |
| 9.4       | Многомерные массивы . . . . .                                      | 70        |
| <b>10</b> | <b>Строки</b>  | <b>74</b> |
| 10.1      | Основные понятия . . . . .   | 74        |
| 10.2      | Особенности . . . . .  | 76        |
| 10.3      | Строки и функции . . . . .   | 77        |
| 10.4      | Работа с символами . . . . .                                       | 82        |
| <b>11</b> | <b>Структуры</b>   | <b>84</b> |
| 11.1      | Оператор typedef . . . . .   | 84        |
| 11.2      | Структуры данных . . . . .   | 84        |
| 11.3      | Работа со структурами . . . . .                                    | 86        |
| <b>12</b> | <b>Файлы</b>   | <b>89</b> |
| 12.1      | Запись . . . . .   | 89        |
| 12.2      | Чтение . . . . .   | 90        |
| <b>13</b> | <b>Распределение памяти</b>  | <b>92</b> |
| 13.1      | void* malloc(size); . . . . .                                      | 92        |
| 13.2      | void* calloc(n, size); . . . . .                                   | 94        |
| 13.3      | void free(ptr); void* realloc(ptr, size); . . . . .                | 95        |
| <b>14</b> | <b>Итоги</b>   | <b>96</b> |
|           | <b>Приложения</b>  | <b>97</b> |
| A         | Полный листинг программы программы терминального калькулятора . .  | 97        |
| B         | Полный листинг программы вычисления среднего арифметического . . . | 98        |
| C         | Полный листинг программы умножения дробей . . . . .                | 98        |

# 1 Введение

## 1.1 От автора

Приветствую, коллеги! Начиная работу над этим документом я не ставил целью написать очередной учебник по языку С или поспорить с трудами Кернигана, Ритчи и Страуструпа. Я думаю, что понял, почему программирование кажется таким сложным: не нашлось ни одного материала, написанного понятным и простым языком. В этом документе я постараюсь не повторить этой досадной ошибки именитых авторов.

|  |
|--|
| Брайан Керниган: « <b>Язык С</b> — инструмент, острый, как бритва: с его помощью можно создать и элегантную программу, и кровавое месиво.» |
|--|

Да, спасибо классикам за предупреждения, постараемся, также, этой работой не создать кровавое месиво.

## 1.2 Инструментарий

В этом документе сознательно пропускается подраздел, посвящённый выбору и настройке инструментария, потому что целью документа не является начало очередного раунда борьбы за право какого бы то ни было компилятора и/или среды разработки называться единственно верным, даже с точки зрения отдельно взятого автора отдельно взятой книги. Обозначим лишь популярные варианты для основной тройки операционных систем (Windows, Linux, Mac OS X). Компиляторы:

- clang;
- GCC/MinGW;
- MSVC.

Остальные, такие как, например, Borland C++ можно считать экзотическими и не пытаться их устанавливать без экономического обоснования. Интегрированные среды разработки и программы для редактирования кода:

- CLion;
- Visual Studio;
- VSCode;
- Qt Creator;

- CodeBlocks;
- Notepad++;
- Sublime Text.

Этот список можно продолжать почти бесконечно, поскольку редактировать код можно в абсолютно любом редакторе, позволяющем сохранять простой текст в файлы.

### 1.3 Немного истории

Первая версия языка C была разработана в 1972м году Деннисом Ритчи для программирования в недавно созданной на тот момент среде UNIX. Язык разрабатывался не государством, а обычными практикующими программистами. В нём сразу были учтены и исправлены все неудобства существовавших на тот момент FORTRAN и PASCAL. Поскольку интерес к языку со временем не пропал, а технологии развивались, появились редакции языка, такие как C99 (1999 год) C11(2011 год). В языке C есть возможность работать с указателями на физические ячейки оперативной памяти компьютера. Конечно, это небезопасно, но при должной квалификации программиста позволяет получить максимально эффективный код, близкий к языку ассемблера и даже машинным кодам конкретного процессора. C является компилируемым процедурным языком со строгой статической типизацией, что позволяет писать максимально безопасный код, и отсеять большую часть ошибок ещё на этапе компиляции проекта. На языке C написано огромное количество программ, библиотек, и даже операционных систем. Какая-бы у Вас ни была установлена операционная система, очень вероятно, что она написана на C. На языке C пишут драйверы для периферийного оборудования, программируют контроллеры для космической аппаратуры, пишут высокоскоростные приложения. Помимо этого, сейчас стремительно набирает популярность такое направление, как «умная техника» и «интернет вещей». Именно из-за этих трендов язык C за последние пару лет снова поднялся в рейтинге TIOBE на лидирующие места. Какой бы язык программирования вы ни изучали, знание языка C нужно потому что языки высокого уровня делают много вещей одной командой, а если Вы хотите не просто стать программистом, а быть хорошим программистом - вы должны понимать, что там, внутри, происходит на самом деле. Знание языка C можно сравнить с умением ездить на автомобиле с механической коробкой передач: коробка-автомат резко снижает порог вхождения в участники дорожного движения, но, зачастую настолько ухудшает качество управления автомобилем, что это приводит к самым печальным последствиям.

## 2 Основные понятия

### 2.1 Компиляция

Прежде, чем говорить о языках программирования и о том, что такое компиляция, как она работает и прочих интересных вещах, нам необходимо познакомиться с понятием, которое будет сопровождать весь курс и в целом всю программистскую жизнь - это понятие имени. Имя - это некий символьный идентификатор (переменная, контейнер) для некоторого числа (числом в свою очередь является адрес ячейки памяти, куда записывается значение). Именованно можно как переменные, так и функции. Простейший пример - запись равенства: `name = 123456`. Различие между именем и числом задает признак числа, в программах для компьютеров признаком числа является первый символ, имя (идентификатор) не должно начинаться с цифры. Таким образом компиляторы однозначно могут определить, что является именем, а что числом. Это отличие накладывает на программиста очевидное ограничение: невозможность создать идентификаторы, начинающиеся с цифр. Также, чтобы не создавать неоднозначности в поведении программы нельзя, чтобы имена в рамках одной программы повторялись. Общий алгоритм работы со всеми компилируемыми языками, в том числе C++ выглядит следующим образом:

1. программист пишет текст программы на каком-либо языке программирования, в нашем случае это C;
2. при помощи программы-транслятора и, зачастую, ассемблера этот текст преобразуется в исполняемые машинные коды, этот процесс, обычно, состоит из нескольких этапов и называется общим словом «компиляция»;
3. исполняемые коды запускаются на целевом компьютере (чаще всего, это сервер или персональный компьютер пользователя).

Применим этот общий алгоритм для написания первой программы. Предполагается, что на данный момент у Вас установлена либо среда разработки, либо текстовый редактор и компилятор по отдельности, чтобы иметь возможность повторить этот и все последующие примеры самостоятельно.

**Первая программа, файл `program.c`** Для написания программы, откроем выбранный текстовый редактор или среду программирования, и напишем следующие строки (важно отличать заглавные и строчные буквы, то есть, например `Int` и `int` - это разные слова, и первое, написанное с заглавной буквы, будет не понято компилятором):

```

1 /*
2  * Project: yet another basic C guide
3  * Author: Ivan I. Ochinnikov
4  * program.c
5  */
6 #include <stdio.h>
7 int main(int argc, char** args) {
8     printf("Hello, World!\n");
9     return 0;
10 }

```

**Запуск компиляции и исполнение программы** В зависимости от выбранного инструментария и ОС процесс компиляции (трансляции) и запуска программы на исполнение может незначительно отличаться, далее будут приведены несколько вариантов. Естественно, что не нужно выполнять их все, а следует выбрать один, который работает именно с Вашим набором инструментов. Трансляция (компиляция):

```

clang -o program program.c
(или) gcc -o program program.c

```

Запуск будет отличаться только для Windows (символами доллара и угловой скобки обозначены приглашения unix-терминала и командной строки windows, соответственно):

```

non-windows $ ./program
windows > .\program.exe

```

Далее в тексте в целях демонстрации будет использоваться запуск в стиле non-windows, а также, будет опускаться демонстрация этапа компиляции (кроме случаев, когда сам процесс компиляции отличается).

## 2.2 Шаблон программы

Как и программы на любом другом языке, программы на языке C имеют ряд обязательных элементов, также называемых шаблоном программы. Рассмотрим эти элементы подробнее на примере только что написанной первой программы (2.1). Некоторые, незначительные аспекты, будут рассмотрены сразу и полностью, комментарии, например, но некоторые будут рассмотрены поверхностно, поскольку являются масштабными, сложными для понимания фундаментальными механизмами языка, которым будут посвящены целые специальные разделы.

**Комментарии** Некоторые среды разработки оставляют в шапке файла комментарии об авторе и дате создания файла. Некоторые команды разработки регламентируют такие комментарии и рекомендуют их написание каждым участником. Комментарий это любой текст, написанный для удобства программиста и игнорируемый компилятором. Комментарии бывают как однострочные, так и многострочные. В редких случаях можно встретить внутрискочные комментарии, но их лучше стараться не использовать, они считаются дурным тоном, поскольку резко снижают читаемость кода.

**Комментарий** - это фрагмент текста программы, который будет проигнорирован компилятором языка.

Очень старые компиляторы допускали только комментарии в стиле `/* xxx */`, сейчас допустим также стиль `// xxx`, завершается такой комментарий концом строки (то есть вся оставшаяся строка, после символов `//` будет проигнорирована компилятором). Комментарии в коде важны, особенно для описания и пояснения неочевидных моментов, но важно соблюсти баланс и не превратить программу в один сплошной комментарий, иногда прерывающийся на работающий код.

**Директивы препроцессора** это такие команды, которые будут выполняться не просто до запуска программы, но даже до компиляции.

Есть мнение, что C/C++ программисты - это не программисты на языке C/C++, а программисты на языке препроцессора используемых ими компиляторов.

В директивах препроцессора подключаются внешние заголовочные файлы, и определяются некоторые абсолютные значения проекта. Обратите внимание, что директивы препроцессора это достаточно сложный инструмент, и использовать его, например, только для определения константных значений - не лучшее архитектурное решение. Для нашего проекта нам понадобится директива `#include <stdio.h>` - эта директива подключит библиотеку стандартного ввода вывода в наш проект, что позволит нам "общаться" с пользователем нашей программы, используя терминал операционной системы (командную строку в терминах Windows)

**Функция `main()`** это точка входа в программу. Программа может состоять из огромного числа файлов и функций, но операционная система как-то должна понять, откуда ей начинать исполнение программы. Такой точкой начала исполнения является функция `main` которая должна быть написана именно так:

```
int main(int argc, char** args)
```

более подробно о функциях, их синтаксисе и аргументах мы поговорим позднее, на данном этапе можно просто запомнить такое (или же упрощённое `int main()`) описание главной функции любой программы на языке C. Далее в фигурных скобках пишется так называемое «тело» программы, то есть именно те операторы, функции и алгоритмы, которые являются программой. По сути, всё наше программирование будет происходить либо в этой функции, либо будет довольно тесно с ней связано.

**Возврат из функции `return`;** это оператор явно завершающий выполнение функции `main` и, соответственно, программы. Все операторы, кроме директив препроцессора, комментариев и описаний тел функций должны заканчиваться точкой с запятой.

**Внимание!** Далее Вы прочитаете тезис, который является значительным упрощением реальной ситуации. Автор пошёл на такое упрощение по двум причинам: во-первых, поскольку в классическом C дела обстояли именно так, а в современных компьютерах ситуация меняется настолько быстро, что никакой текст не сможет оставаться актуальным, и во-вторых, поскольку целью данного документа не является детальное описание архитектур современных операционных систем.

Поскольку программа написанная на языке C работает на одном уровне с операционной системой, а не в средах виртуализации, как это происходит в Java, например, она должна сообщить операционной системе, что она отработала нормально. Это делается посредством возврата в качестве результата работы программы кода ноль. В нашем случае, оператор `return` сообщает код 0, говорящий об успешности завершения работы программы. Такой возвратный код - исторически сложившаяся договорённость между программистами: ненулевой код означает аварийное завершение программы и сообщает системе, что программа завершена некорректно и необходимо дополнительно и явно освободить занятые ею ресурсы.

## 3 Базовый ввод-вывод

### 3.1 Форматированный вывод

Общение с пользователем на чистом C происходит через консоль. Для того, чтобы выводить какую-либо информацию для чтения пользователем - используется функция `printf()`; предназначенная для форматированного вывода некоторого текста в консоль. Функция описана в заголовке `stdio.h`, поэтому мы и включили данный заголовок в нашу программу. Какого рода форматирование применяется при выводе строк в консоль? Существуют два основных инструмента придания выводу необходимого вида: экранированные последовательности (*escape sequences*) и заполнители (*placeholders*).

**Экранированная последовательность** — это буква или символ, написанные после знака обратного слэша (`\`), которые при выполнении программы будут на что-то заменены. Самые часто используемые это:

- `\'` - одинарная кавычка;
- `\"` - двойная кавычка;
- `\?` - вопросительный знак;
- `\\` - обратный слэш;
- `\0` - нулевой символ;
- `\b` - забой (*backspace*);
- `\f` - перевод страницы - новая страница;
- `\n` - перевод строки - новая строка;
- `\r` - возврат каретки;
- `\t` - горизонтальная табуляция;
- `\v` - вертикальная табуляция;
- `\nnn` - произвольное восьмеричное значение;
- `\xnn` - произвольное шестнадцатеричное значение;
- `\unnnn` - произвольное Юникод-значение.



- %% - символ процента;
- %i (%d) - целые числа (integer, decimal);
- %ld (%li) – целые числа (long int);
- %s - для вывода строк;
- %c – для вывода символов;
- %p - для вывода указателей;
- %f – для вывода чисел с плавающей точкой;
- %lf – для вывода чисел с плавающей точкой удвоенной точности;
- %x (%X) - беззнаковое целое в шестнадцатеричном виде.

Как видно, существуют заполнители для всех основных типов данных и для экранирования самого символа начала заполнителя. Заполнители можно и нужно комбинировать в строках как между собой, так и с экранированными последовательностями. Умение работать с заполнителями пригодится не только в консоли, но и при формировании локализованных сообщений в более сложных приложениях, в том числе на других языках.

```

1  int a = 50;
2  printf("%d\n", a);
3  printf("%5d\n", a);
4  printf("%05d\n", a);
5  printf("%.2f\n", 5.12345);
6  printf("Placeholders are \"%5d%%\" of formatting\n", a);
7

```

Так первый оператор выведет просто число. Второй это же число, но оставив для его отображения пять пробельных символов (два окажутся заняты разрядами числа 50, и ещё три останутся свободными, слева от числа). Третий оператор форматированного вывода впишет число в пять отображаемых символов, но заполнит пустоту нулями (запись с лидирующими нулями, leading zeroes). Четвёртый осуществит вывод числа с плавающей точкой, ограничив дробную часть двумя отображаемыми символами, при этом важно, что не произойдёт математического округления, символы просто не отобразятся, такое отображение часто используется для демонстрации денежных значений в долларах и центах, рублях и копейках, и пр. Последний же оператор выведет на экран надпись, информирующую о том, что заполнители - это 50% форматирования:

```

$ ./program
50
    50
00050
5.12
Placeholders are "    50%" of formatting
$

```

Для заполнителей `%d`, `%i`, `%f` часто используются дополнительные параметры, такие как количество знаков после запятой, например, `%.2f` или минимальное количество знаков для отображения целого числа `%5d`. Также в пользу оператора форматированного вывода говорит тот факт, что, например, в C++ стандартный вывод в консоль осуществляется с помощью команды `std::cout`, которая не поддерживала форматирование строк вплоть до принятия стандарта C++20.

## 3.2 Форматированный ввод

Поговорив о выводе в консоль нельзя не сказать о пользовательском вводе данных. Один из способов пользовательского ввода данных в программу - это использование функции `scanf()`. Предложим пользователю ввести некоторое число:

```

1  int input;
2  printf("Please, enter a number: ");
3  scanf("%d", &input);
4

```

Функция `scanf()` – это функция форматированного ввода. Принцип её работы очень похож на принцип работы функции `printf()`; В двойных кавычках мы указываем в виде заполнителя тип переменной, которую ожидаем от пользователя, а в качестве дополнительного аргумента указываем адрес той переменной, в которую хотим записать введённые пользователем данные. Получается процесс прямо противоположный выводу. В этой функции можно использовать все те же заполнители, что и при выводе, поэтому пользователь может ввести как целые числа, так и символы, строки и числа с плавающей точкой.

```

1  printf("You entered %d, let's double it: %d\n", input, input * 2);

```

Выведем в консоль изменённое число, введённое пользователем, чтобы удостовериться, что всё работает. В результате запуска программы, консоль застынет в ожидании

пользовательского ввода. Пользователь сообщает консоли (терминалу операционной системы) об окончании ввода нажатием клавиши Enter:

```
$ ./program  
Please, enter a number: 50  
You entered 50, let's double it: 100  
$
```

Функция форматированного ввода позволяет не только приводить пользовательский ввод к нужному типу данных, но и считывать разные сложные пользовательские вводы из множества переменных, разделённых символами.

## 4 Переменные и типы данных. Базовые манипуляции с данными

### 4.1 Переменные в программе на языке C

Это некие *именованные контейнеры*, тип которых строго описан при их создании, каждый из которых может содержать одно и только одно значение в единицу времени. Названия или имена переменных не могут начинаться с цифр и спецсимволов, а также не должны повторяться (2.1).

**Идентификатор** переменной - это её имя, которое для программы не существует без привязки к типизации, то есть для объявления переменной мы пишем её тип и название, что вместе составляет идентификатор. По идентификатору переменной мы можем записать в неё значение, прочитать текущее значение, узнать адрес хранения этого значения, и т.д. **Литерал** - это число или строка, которые мы пишем в тексте явно. Литералу нельзя присвоить значение, литерал это и есть значение. Литерал (ни строковый, ни числовой) нельзя изменить. Если изменить какой-то литерал, то это будет уже другой литерал, явно изменённое в коде значение. Также есть термины `lvalue` и `rvalue`. Если очень сильно упрощать, то их можно отождествить с идентификатором и литералом: `lvalue` - это то, куда присваивается, `rvalue` - это то, что присваивается

Переменные делятся на целочисленные, символьные, указатели и числа с плавающей точкой (англ. floating point, дробное число). Все, кроме указателей и символьных переменных бывают как знаковыми так и беззнаковыми. То есть в знаковых самый старший бит в двоичной записи этих переменных отводится под определение, является ли число отрицательным, или положительным, в беззнаковых все биты используются для записи числа, что увеличивает его диапазон возможных значений, но позволяет записать только положительные числа. В классическом C нет булевого типа, вместо него используется целое число и значения нуля для **лжи** и **любое** другое число для **истины**, обычно это единица. Об указателях и булевой алгебре мы будем подробно говорить в одном из последующих разделов.

| Тип                        | Пояснение  | Спецификатор формата                                   |
|----------------------------|--|--|
| char                       | Целочисленный, самый маленький из адресуемых типов, диапазон: [-128, +127]   | %c   |
| short<br>short int         | Тип короткого целого числа со знаком, диапазон: [-32 768, +32 767]   | %hi  |
| int                        | Основной тип целого числа со знаком, диапазон: [-2 147 483 648, +2 147 483 647]  | %i или %d  |
| long<br>long int           | Тип длинного целого числа со знаком, диапазон: [-2 147 483 648, +2 147 483 647]  | %li или %ld  |
| long long<br>long long int | Тип двойного длинного целого числа со знаком, диапазон: [-9 223 372 036 854 775 808, +9 223 372 036 854 775 807]       | %lli   |
| float                      | Тип вещественного числа с плавающей запятой (одинарной точности)   | %f (автоматически преобразуется в double для printf()) |
| double                     | Тип вещественного числа с плавающей запятой (двойной точности)   | %f(%F)<br>(%lf(%LF)<br>для scanf())<br>%g %G %e %E     |
| long double                | Тип вещественного числа с плавающей запятой, ставящийся в соответствие формату повышенной точности с плавающей запятой | %Lf %LF %Lg<br>%LG %Le %LE                             |

1: Основные типы данных в языке C

**Символьный тип** не такой простой, как может показаться на первый взгляд. Если вкратце, то в переменной типа `char` хранится число, которое можно интерпретировать как символ. По умолчанию тип знаковый, то есть может содержать значения от -128 до +127, но символы в таблице ASCII<sup>1</sup>, что совершенно логично, имеют только положительные индексы, поэтому в читаемый текст в стандартном C можно превратить только латинский алфавит и некоторый набор знаков и символов, находящиеся на первых 128-ми местах в этой таблице. Также можно явно указать компилятору, что мы хотим использовать эту переменную как беззнаковую, для этого используется ключевое слово `unsigned`, что позволит нам хранить только положительные числа гораздо больших значений. Например для переменной типа `unsigned char` это будут значения от 0 до 255, а для переменной типа `unsigned int` можно можно хранить значения от 0 до +4.294 миллиардов с какими-то копейками. В более поздних редакциях языка были

<sup>1</sup>American standard code for interaction interchange

утверждены типы `long long` и другие, для хранения 64-х разрядных целых чисел.

```
1 unsigned char symbol = 75;  
2 printf("75 stands for: %c\n", symbol);  
3
```

Соответственно, программа выше выведет в терминал информацию о том, какой именно символ в таблице ASCII соответствует позиции 75, по большому счёту, можно было обойтись и без ключевого слова `unsigned`, но когда мы говорим именно о символах, а не просто о минимальных по занимаемому размеру целых числах, принято выделять их:

```
$ ./program  
75 stands for: K  
$
```

**Числа с плавающей точкой (дробные)** представлены двумя типами: четырёхбайтный `float` и восьмибайтный `double` (также называемый `long float`). Хранятся в памяти в неявном виде, а разделённые на мантиссу экспоненту и знак, что делает их одними из самых сложных в работе<sup>2</sup>.

Важно, что компилятор, при работе с текстом программы считает все литералы в коде числами типа `double`, а значит код вида `float var = 0.123;` будет отмечен компилятором как неверный, поскольку компилятор не в состоянии положить восемь байт информации в четырёхбайтную переменную. То есть, если мы хотим инициализировать переменную типа `float`, то нам нужно специальным образом пометить литерал: `float var = 0.123f;`. Символ `f` в конце явно указывает на то, что литерал имеет тип `float`.

При работе с числами с плавающей точкой нужно обращать особенное внимание на тип переменной, поскольку сравнение внешне одинаковых чисел разных типов с почти стопроцентной вероятностью даст ложный результат, в отличие от сравнения простых целых чисел. Об операциях и операторах сравнения мы поговорим позже, сейчас просто приведём наглядный пример того, как это работает.

<sup>2</sup>Здесь имеется ввиду внутренняя работа самого компьютера, а не работа с такими типами с точки зрения программиста. Именно из-за сложности, многие старые процессорные архитектуры не имели возможности работать с переменными такого типа

```

1 float real = 5,345f; // 4 bytes
2 double realdouble = 5,345; // 8 bytes
3 printf("float and double: %d\n", real == realdouble);
4
5 int a = 10;
6 int b = 10;
7 printf("integers: %d\n", a == b);
8

```

Запустим код, выводящий результаты сравнений и убедимся в том, что прямое сравнение дробных чисел допускать нежелательно. Обычно, если есть необходимость в сравнении дробных чисел применяют сравнение с некоторой допустимой точностью, например, до третьего или пятого знака после запятой. О таких сравнениях мы поговорим позднее.

```

$ ./program
float and double: 0
integers: 1
$

```

Иногда бывает важным отметить, что рассмотрение языка C для решения задач на уровне персональных компьютеров может отличаться от рассмотрения языка C для решения прикладных задач аппаратуры (микроконтроллеров и микропроцессорных операционных систем). Так может оказаться важным, что тип данных `int` не всегда является тридцатидвухразрядным, а его размер аппаратно-зависимый. Для разрешения подобных неточностей в более новых стандартах языка вводятся и принимаются гарантированно фиксированные по размеру переменные

**Тип данных - указатель.** Как было сказано - переменная это именованный контейнер. У каждого такого контейнера есть свой собственный адрес в оперативной памяти. Язык C позволяет узнать этот адрес и работать с ним. Оператор взятия адреса это знак амперсанд (&), написанный перед именем переменной. То есть у любой переменной всегда есть значение и адрес где это значение хранится (немного подробнее на стр. 58). Для вывода в консоль адреса используется специальный заполнитель - `%p`.

```

1 printf("Variable a has value: %d \n", a);
2 printf("Variable a stored at: %p \n", &a);
3

```

При неоднократном запуске кода можно обратить внимание, что первые цифры в

адресе всегда остаются неизменными, а последние меняются редко, это связано с тем, что в современных операционных системах пользователю для его работы чаще всего выделяется некоторое адресное пространство, которое потом просто переиспользуется и перезаписывается. При запуске точно такого же кода на Вашем компьютере, Вы увидите, что адрес хранения переменной наверняка будет отличаться, это нормально.

```
$ ./program
Variable a has value: 10
Variable a stored at: 0x7ffe6aa136cf
$
```

## 4.2 Базовая арифметика

Раз уж в предыдущем разделе мы коснулись арифметических выражений, поговорим немного об арифметике.

**Простые арифметические операции.** В языке C поддерживаются все базовые арифметические операции, такие как сложение, вычитание, умножение, деление. Операции бинарной арифметики (булевой алгебры), такие как И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ, СДВИГИ. А также все вышеперечисленные операции с последующим присваиванием в первую переменную. Для начала, инициализируем переменную типа `int` значением, например, 70, и выведем ее в консоль. Мы можем производить с этой переменной все привычные базовые арифметические манипуляции, ровно также, как мы можем производить эти манипуляции с литералом, который ей присваивается:

```
1  int variable = 70;
2  printf("The variable = %d\n", variable);
3  printf("The variable = %d\n", variable + 10);
4  variable = variable + 50;
5  printf("The variable = %d\n", variable);
6  variable = 123 + 50 * 12;
7  printf("The variable = %d\n", variable);
8
```

То есть, нам доступны операции сложения, умножения, вычитания и деления. Как видно в результате работы программы, есть прямая зависимость между действительным значением переменной и порядком присваивания в неё значения, так в третьем выводе значение равно 120, то есть  $70 + 50$ , а значит во втором выводе присваивания нового значения  $70 + 10$  в переменную `variable` не произошло.

```
$ ./program
The variable = 70
The variable = 80
The variable = 120
The variable = 723
$
```

**Оператор деления** заслуживает особенного внимания, поскольку у него есть два режима работы, отличающие этот оператор от привычной нам арифметики: если мы производим операции с целыми числами такими как `int`, `short` или `char` оператор деления всегда будет возвращать только целые числа, **отбросив дробную часть**. Это происходит из-за оптимизаций компилятора, то есть если операнды - это целые числа, то и результатом по мнению компьютера может быть только целое число, а из-за строгости типизации мы не можем положить в целочисленную переменную значение с плавающей точкой.

Таким образом, отслеживание точности вычислений полностью возлагается на плечи программиста. Важно, что компилятор нам в этом помогает, хоть и не делает всю работу за нас. Компилятор, при преобразовании операций автоматически приводит типы операндов к наиболее подходящему и широкому. То есть, если мы, например, складываем два числа `int` и `char`, то `char` будет автоматически расширен до `int`, потому что максимальное значение `char` точно поместится в переменную типа `int`, а максимальное значение `int` точно никак не сможет поместиться в переменную с типом `char`. Точно также если умножить `int` и `float`, то `int` будет преобразован во `float` по той же причине - `int` совсем никак не умеет работать с плавающей точкой, а `float` вполне может содержать число без дробной части. Из-за этого языки C/C++ считаются слабо типизированными.

```
1   variable = 10;
2   variable = variable / 3;
3   printf("The variable = %d\n", variable);
4
5   float var = 10;
6   var = var / 3;
7   printf("The var = %f\n", var);
8
```

В примере выше, обратите внимание, что переменная `variable` не была инициализирована, а ей просто было присвоено значение. Это сделано потому, что данный участок кода является продолжением кода примеров простой арифметики из предыду-

шего параграфа. При использовании оператора деления с целочисленными операндами теряется точность вычислений, что недопустимо.

Чтобы оператор деления отработал в не целочисленном режиме, нужно, чтобы хотя бы один операнд был не целочисленным.

Чаще всего целочисленные переменные используют в качестве счётчиков, индексов и других вспомогательных переменных, поэтому математические операции с ними весьма распространены.

```
$ ./program
The variable = 3
The var = 3.333333
$
```

**Деление по модулю.** Также особенного внимания заслуживает оператор получения остатка от деления, иногда называемый оператором взятия по модулю. Записывается как символ % и возвращает остаток от целочисленного деления первого числа на второе:

```
1  int remain = variable % 5;
2  printf("Division remainder of %d by %d: %d\n", variable, 5, remain);
3  variable = variable + 50;
4  printf("The variable = %d\n", variable);
5  variable += 50;
6  printf("The variable = %d\n", variable);
7
```

Любые арифметические операции можно выполнить с последующим присваиванием в первую переменную. То есть это означает, что запись вида `variable = variable + 50`; можно сократить до `variable += 50`; и запустив следующий код мы можем убедиться, что начальное значение переменной увеличилось сначала на 50, а затем ещё на 50.

```
$ ./program
Division remainder of 3 by 5: 3
The variable = 53
The variable = 103
$
```

**Инкремент и декремент.** Так, бегло рассмотрев арифметику в языке С нельзя не упомянуть об операторах увеличения и уменьшения значения переменной на единицу с последующим присваиванием. Они называются операторами инкремента (++) и декремента (--). Это унарный оператор, поэтому записывается со своим операндом строго без пробелов: `variable++`; и редко используется как самостоятельный оператор на отдельной строке. У операторов инкремента и декремента есть два вида записи: префиксный и постфиксный. Их отличает время применения текущего значения и его изменения. При постфиксной записи, сначала происходит применение текущего результата, а затем его изменение, а при префиксной записи - сначала изменение, а затем применение. Например:

```
1   variable = 50;
2   printf("1. Postfix increment: %d\n", variable++);
3   printf("1. Next line of code: %d\n", variable);
4   variable = 50;
5   printf("2. Prefix increment: %d\n", ++variable);
6   printf("2. Next line of code: %d\n", variable);
7
```

В результате выполнения этого кода мы можем видеть на экране следующий результат:

```
$ ./program
1. Postfix increment: 50
1. Next line of code: 51
2. Prefix increment: 51
2. Next line of code: 51
$
```

### 4.3 Булева алгебра и двоичные вычисления

**Двоичная система счисления** представляет особенный интерес для области информационных технологий, поскольку вся электроника работает по принципу «есть напряжение или нет напряжения», единица или ноль. Все числа из любых систем счисления в результате преобразуются в двоичную и представляются в виде набора единиц и нулей. Так для записи десятичного числа 116 используется двоичная запись 1110100. Преобразование из системы счисления с большим основанием в систему счисления с меньшим основанием производится последовательным делением исходного числа на основание системы счисления и записи остатков такого деления в младшие разряды.

Например:

$$\frac{116}{2} = \frac{58(0)}{2} = \frac{29(0)}{2} = \frac{14(1)}{2} = \frac{7(0)}{2} = \frac{3(1)}{2} = \frac{1(1)}{2} = 1 < 2$$

В этом примере полученные остатки от деления записаны в скобках и можно обратить внимание на то, что они полностью повторяют запись числа 116 показанную ранее в зеркальном отражении. Обратное преобразование - это последовательное умножение разрядов числа на величину каждого разряда с их аккумулярованием к общему результату:

$$\begin{aligned} 1110100 &= 0 * 2^0 + 0 * 2^1 + 1 * 2^2 + 0 * 2^3 + 1 * 2^4 + 1 * 2^5 + 1 * 2^6 = \\ &= 0 * 1 + 0 * 2 + 1 * 4 + 0 * 8 + 1 * 16 + 1 * 32 + 1 * 64 = \\ &= 4 + 16 + 32 + 64 = 116 \end{aligned}$$

Поскольку двоичная система счисления является основной для компьютерной техники, помнить, например, значения степеней двойки - обычно, хорошее подспорье в работе.

**Булева алгебра** это один из базовых, но вместе с тем один из самых мощных инструментов в программировании. Двоичные вычисления выполняются быстрее десятичных, поскольку являются естественными для цифровой техники. В бинарной алгебре используются операторы И (&), ИЛИ (|), НЕ (~), ИСКЛЮЧАЮЩЕЕ ИЛИ (^) и операции СДВИГА влево (<<) и вправо(>>). Работают эти операторы относительно разрядов двоичного представления чисел, где истина – это единица, а ложь - это ноль.

Разница между логическими и арифметическими бинарными операторами в представлении операндов: логические оперируют числовыми литералами и переменными целиком, а арифметические числами поразрядно. Работу логических операторов мы рассмотрим в следующем разделе.

Условия истинности двоичных арифметических операторов следующие:

- оператор И возвращает единицу только когда оба операнда единицы;

| операнд | операнд | результат |
|---------|---------|-----------|
| 0       | 0       | 0         |
| 0       | 1       | 0         |
| 1       | 0       | 0         |
| 1       | 1       | 1         |

- оператор ИЛИ возвращает единицу когда хотя бы один из операндов единица;

| операнд | операнд | результат |
|---------|---------|-----------|
| 0       | 0       | 0         |
| 0       | 1       | 1         |
| 1       | 0       | 1         |
| 1       | 1       | 1         |

- оператор НЕ возвращает единицу когда операнд равен нулю;

| операнд | результат |
|---------|-----------|
| 0       | 1         |
| 1       | 0         |

- оператор ИСКЛЮЧАЮЩЕГО ИЛИ возвращает единицу когда операнды различаются.

| операнд | операнд | результат |
|---------|---------|-----------|
| 0       | 0       | 0         |
| 0       | 1       | 1         |
| 1       | 0       | 1         |
| 1       | 1       | 0         |

На основе этих знаний мы можем для примера написать программу, меняющую местами значения переменных без использования третьей, вспомогательной и быть уверенными, что переполнения переменных не произойдёт, как это могло бы произойти, например, при использовании сложения и обратного вычитания. Объявим две переменных `a` и `b`, присвоим им значения и выведем их в консоль. Также подготовим вывод изменённых значений `a` и `b` в консоль:

```

1  char a = 11;
2  char b = 15;
3  printf("a = %d, b = %d\n", a, b);
4  // here will be the swapping algorithm
5  printf("a = %d, b = %d\n", a, b);
6

```

Далее, напишем некую конструкцию, которая при детальном изучении не представляет из себя никакой магии. В переменную `a` нужно будет записать результат вычисления  $a \wedge b$ , в переменную `b` нужно будет записать результат вычисления  $b \wedge a$  и наконец в переменную `a` нужно будет записать результат вычисления  $a \wedge b$ , в коде ниже будет приведена сразу сокращённая запись:

```
1   a ^= b;  
2   b ^= a;  
3   a ^= b;  
4
```

Нужно сразу оговориться, что этот алгоритм может некорректно работать с одинаковыми и отрицательными числами, это будет зависеть от компилятора, поэтому, если включать этот алгоритм в состав более сложных, лучше осуществлять дополнительные проверки. Вывод этой конкретной программы будет следующим:

```
$ ./program  
a = 11, b = 15  
a = 15, b = 11  
$
```

Дополнительно, для написания этого документа был проведён ряд тестов:

```
Test project ~/Documents/c-basic/build  
1/7 Test #1: Swap.TwoPosNumbers ..... Passed 0.00 sec  
2/7 Test #2: Swap.SamePosNumbers .... Passed 0.00 sec  
3/7 Test #3: Swap.OneNegNumber ..... Passed 0.00 sec  
4/7 Test #4: Swap.TwoNegNumbers ..... Passed 0.00 sec  
5/7 Test #5: Swap.SameNegNumbers .... Passed 0.00 sec  
6/7 Test #6: Swap.BareOverflow ..... Passed 0.00 sec  
7/7 Test #7: Swap.OverflowNumbers ...***Failed 0.00 sec
```

```
86% tests passed, 1 tests failed out of 7  
Total Test time (real) = 0.04 sec
```

Ожидаемо, не прошёл тест, в котором присутствовало переполнение переменной, этот случай также был отмечен предупреждением компилятора о том, что программист пытается присвоить переменной значение, большее, чем переменная способна вместить.

Здесь был преднамеренно использован тест с провальным результатом, для более явной демонстрации происходящего внутри алгоритма, и потому что нам не нужна дальнейшая компиляция продакшн кода. Обычно, тест-кейсы с ожидаемым провалом пишутся с инверсией проверки, то есть, если мы ожидаем, что некоторые значения не будут равны эталонным, необходимо проверять значения на неравенство, таким образом все тест-кейсы пройдут успешно.

Рассмотрим происходящее для приведённого примера кода пошагово: оператор ИСКЛЮЧАЮЩЕГО ИЛИ выполняется следующим образом: результат будет равен 1 если операнды (в данном случае, разряды двоичного представления числа) различаются и 0 если они совпадают. Изначально имеем две переменных, *a* и *b* - число 11 типа *char* (в двоичном представлении это 00001011), и число 15 (это 00001111). В коде ниже можно наглядно рассмотреть работу оператора ИСКЛЮЧАЮЩЕГО ИЛИ:

```
// a = 11    (00001011)
// b = 15    (00001111)
a = a ^ b; //00000100
```

После выполнения первого оператора в переменную *a* будет положено промежуточное число 00000100 – это цифра 4, а в переменной *b* останется число 15 (00001111). Ко второму действию мы приходим с начальными значениями: *a* = 4 (00000100), *b* = 15 (00001111), производим операцию ИСКЛЮЧАЮЩЕГО ИЛИ с последующим присваиванием в переменную *b* и получаем 00001011 – т.е. *b* = 11 (00001011).

```
// b = 15    (00001111)
// a = 4      (00000100)
b = b ^ a; //00001011
```

И после выполнения третьего оператора ИСКЛЮЧАЮЩЕГО ИЛИ в переменную *a* будет положено значение 00001111 – это цифра 15.

```
// a = 4      (00000100)
// b = 11     (00001011)
a = a ^ b; //00001111
```

**Операции сдвига** бывают логические, арифметические и циклические. В языке C реализован логический сдвиг, то есть недостающие разряды при сдвиге заполняются нулями, а выходящие за пределы хранения переменной теряются без возможности восстановления. Итак, допустим, что нам нужно переменную *a* сдвинуть влево на 3 бита, на самом деле это означает что мы переменную *a* умножим на  $2^3$ . В примере ниже, переменную *b* мы сдвинем вправо на 2 бита, это означает, что мы переменную *b* разделим на  $2^2$ , при этом важно упомянуть, что будет произведено целочисленное деление.

**Сдвиг влево** числа  $k$  на  $n$  - это *умножение*  $k * 2^n$ . **Сдвиг вправо** числа  $k$  на  $n$  - это *целочисленное деление*  $\frac{k}{2^n}$ .

Это тоже самое что записать  $a*8$  и  $b/4$ . Просто на маломощных компьютерах выполниться это гораздо быстрее. Бинарная алгебра это большая и интересная тема, на которую написано немало статей и даже книг, но подробное её изучение выходит далеко за рамки знакомства с синтаксическими основами языка. Также важно помнить, что бинарная алгебра не работает с дробными числами по причине их сложного (4.1) хранения.

```
1  a = 15; //00001111
2  b = 11; //00001011
3  printf("a = %d", a);
4  a = a << 3; // 15 * 8
5  printf("a = %d", a); //a = 120; //01111000
6  printf("b = %d", b);
7  b = b >> 2; // 11 / 4
8  printf("b = %d", b); //b = 2; //00000010
9
```

Применять бинарную алгебру можно и в больших проектах, работающих со сложными высокоуровневыми абстракциями. Помимо этого важно помнить, что поддержка бинарных операций есть в подавляющем числе языков программирования. Используя бинарную алгебру можно создавать оптимальные протоколы передачи данных и/или алгоритмы хранения и обработки.

```
$ ./program
a = 15
a = 120
b = 11
b = 2
$
```

Тема битовых операций постепенно теряет свою актуальность, в связи с развитием технологий, скоростей, объёмов. Скорее всего, битовые операции в ближайшем будущем перейдут в разряд узкоспециальных знаний и будут применяться только при программировании микроконтроллеров, несмотря на то, что работа с двоичным представлением чисел открывает перед программистом широкий простор к оптимизации и ускорению собственного кода.

## 5 Условия, блоки кода, видимость

### 5.1 Условный оператор

`if()` пожалуй, самый часто используемый в любом языке программирования, в том числе и в языке C оператор. Оператор `if()` позволяет программе принять решение о выполнении или невыполнении того или иного действия в зависимости от текущего состояния. В случае, если условие в круглых скобках выполнится, выполнится и последующий код, который чаще всего пишется в фигурных скобках. Если условие в круглых скобках не выполнится, то все операторы внутри идущих следом фигурных скобок будут проигнорированы.

Например, зададим пользователю вопрос, хочет ли он, чтобы его поприветствовали, для этого опишем переменную `char answer`, которая будет хранить ответ пользователя в виде символа и спросим у пользователя в терминале, хочет ли он, чтобы мы его поприветствовали, выведем на экран строку с приглашением. Далее при помощи уже знакомой нам функции `scanf()`; считаем ответ пользователя в переменную, и, в зависимости от пользовательского ввода, программа либо поприветствует пользователя, либо нет, это решение будет принято с помощью оператора `if()`.

```
1  char answer;
2  printf("do you want me to salute you (y/n)? ");
3
4  scanf ("%s", &answer);
5  if (answer == 'y') {
6      printf("Hello, user");
7  }
8
```

**else** Зачастую складываются ситуации, когда нужно выполнить разные наборы действий, в зависимости от результата проверки условия. Для таких случаев используется дополнение к оператору `if()` - оператор `else`, в котором описывается последовательность действий выполняемая в случае если условие в круглых скобках дало ложный результат. Код немного изменится, не будем приводить повторяющиеся части, взаимодействия с пользователем, сконцентрируемся на условном операторе:

Как вы видите, в зависимости от того что ввел пользователь мы реализуем ту или иную ветку оператора `if-else`. Конструкция `if-else` является единым оператором выбора, то есть выполнив код в фигурных скобках после `if` программа не станет выполнять код в `else`, и наоборот.

```

1     if (answer == 'y') {
2         printf("Hello, user");
3     } else {
4         printf("I didn't want to salute you anyway");
5     }
6

```

**else if()** Множественный выбор при помощи оператора `if` можно осуществить используя конструкцию `if-else if-else`. Данное усложнение также будет являться единым оператором выбора. Добавим в нашу конструкцию еще одно условие и опишем поведение для ответа «да» и ответа «нет». В этом примере оператором `else` будет непонимание программы того что ввел пользователь. Выведем в консоль надпись «Я не могу понять Ваш ввод».

```

1     if (answer == 'y') {
2         printf("Hello, user");
3     } else if (answer == 'n') {
4         printf("I didn't want to salute you anyway");
5     } else {
6         printf("I can't understand your input");
7     }
8

```

Операторов `else if` в одном операторе выбора может быть сколько угодно, в отличие от оператора `if` и оператора `else` которых не может быть больше одного.

```

$ ./program
do you want me to salute you (y/n)? y
Hello, user
$ ./program
do you want me to salute you (y/n)? n
I didn't want to salute you anyway
$ ./program
do you want me to salute you (y/n)? x
I can't understand your input
$

```

**Тернарный оператор.** Для короткой или внутрискочной записи условного оператора, а также для присваивания переменных по условию можно использовать **тернарный оператор**, также называемый оператором условного перехода и записываемый с помощью следующего синтаксиса: `(условие) ? истина : ложь`. Например, создадим

три целочисленные переменные `a`, `b`, `c` и зададим двум из них какие-нибудь начальные значения, допустим `a = 10` и `b = 15`. Поставим себе задачу: присвоить переменной `c` наименьшее из значений `a` или `b`. Если мы будем использовать только что изученный нами оператор `if-else` у нас должен получиться такой код:

```
1   int a = 10;
2   int b = 15;
3   int c;
4   if (a > b) {
5       c = b;
6   } else {
7       c = a;
8   }
9
```

Запись условного оператора можно значительно сократить, поскольку в теле оператора происходит выбор: какое значение присвоить в `c` по условию в круглых скобках. Условие оставляем, а `a` и `b` переносим в секции истины и лжи, соответственно. Получим запись вида:

```
1   int a = 10;
2   int b = 15;
3   int c = (a > b) ? b : a;
4
```

которая будет обозначать, что в случае если `a > b`, в переменную `c` запишется значение `b`, и наоборот если `b > a`, то в переменную `c` запишется значение `a`. Также тернарный оператор можно использовать для удобного форматированного вывода, например опишем функцию `printf()`; которая будет печатать нам строку, и в зависимости от условия, это будет `"true"` либо `"false"`:

```
1   printf("%s", (1 > 0) ? "true" : "false");
```

Проверим как это работает, и в результате видим `true`, потому что единица действительно больше нуля.

```
$ ./program
true
$
```

**Вложенные условия и сокращения** Внутри фигурных скобок конструкций `if()` находится код программы, поэтому там могут находиться и другие условные операторы.

Условия, расположенные таким образом называются вложенными. Никаких ограничений на использование вложенных условий в языке С нет. В примере ниже показано, что условия всегда выполняются (единица в круглых скобках будет означать, что условие всегда истинно), а комментариями с многоточием показано, где может располагаться код программы.

```
1     if (1) {
2         // operators
3         if (1) {
4             // operators
5         }
6         // operators
7     }
```

Также важно отметить, что если после условных операторов следует только один оператор языка, как в примере ниже, то использование фигурных скобок не обязательно, хотя и считается хорошим тоном писать их всегда:

```
1     // one operator condirion
2     if (1) {
3         // single operator
4     }
5
6     // also one operator condition
7     if (1)
8         // single operator
9
10    // another one operator condition
11    if (1)
12        // single operator
13    else
14        // single operator
15
```

При использовании такой записи можно легко допустить ошибку: забыть о необходимости объединения кода фигурными скобками, и предполагать, что несколько операторов могут выполняться по условию без фигурных скобок.

Часто это заблуждение приводит к трудноуловимым ошибкам в коде, когда программа компилируется, запускается, но работает не так, как следует.

## 5.2 Операции сравнения

**Арифметическое сравнение** это знакомые и привычные нам со школы операторы «больше» (>), «меньше» (<), «больше или равно» (>=), «меньше или равно» (<=), а также в языке C присутствуют в виде отдельных операторов «проверка на равенство», которая записывается в виде двух знаков равенства (==), и «проверка на неравенство», которая записывается в виде восклицательного знака и символа равенства (!=). Возвращают истину, когда выполняются соответствующие названиям условия и ложь, когда условия не выполняются, что очевидно. Единственное исключение, которое было оговорено ранее (4.1), сравнение разнотипных нецелочисленных значений должно осуществляться через сравнение допустимой дельты этих значений, например:

```
1 float f = 0.5f;
2 double d = 0.5;
3 fouble diff = (f > d) ? f - d : d - f;
4 if (diff < 0.00001) {
5     // useful code here
6 }
7
```

**Логические операторы** Их три: это И (&&), ИЛИ (||), НЕ (!). В отличие от арифметических двоичных операторов - логические возвращают истину или ложь т.е. в случае языка C - 1 либо 0 и работают с операндами слева и справа целиком, а не поразрядно. В этом легко убедиться, попытавшись вывести на экран результат сравнения с заведомо неверным форматированием и получив ошибку, говорящую о том, что компилятор ожидал от нас число, а мы хотим его отформатировать в строку.

```
1 printf("%s\n", 1 == 1);
```

Некоторые компиляторы выдают ошибку на этапе компиляции, некоторые компилируют такой код, но программа не сможет выполниться и выдаст ошибку **Segmentation fault**, то есть ошибку доступа к памяти (попытка обратиться к недоступной памяти или попытка обратиться к памяти неподобающим образом). В этой конкретной ситуации, мы попытаемся интерпретировать как строку часть памяти, которая находится по адресу 1, что находится далеко за пределами доступа программы. Это поведение (ошибка компиляции или ошибка времени выполнения) зависит от большого количества факторов, таких как тип операционной системы, тип и версия компилятора, версия используемого стандарта языка.

Отдельного внимания заслуживает применение оператора поразрядного (арифметического) ИСКЛЮЧАЮЩЕГО ИЛИ в качестве логического. В случае такого применения

оператор ИСКЛЮЧАЮЩЕГО ИЛИ, фактически, дублирует сравнение на неравенство, это легко объяснить, проведя анализ происходящего с числами при таком сравнении:

```
1 int a = 11; //00001011
2 int b = 11; //00001011
3 // ^ 00000000
4 if (a ^ b) {
5     printf("numbers are not equal\n");
6 }
7
```

Данный код внутри фигурных скобок оператора `if()` никогда не выполнится, поскольку в С оператор сравнения работает с числами, интерпретируя ноль как ложь, а любое ненулевое значение как истину, мы наблюдаем, что побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ - это тоже самое, что проверка на неравенство.

- оператор И (`&&`) возвращает истину только когда оба операнда истинны;

| операнд | операнд | результат |
|---------|---------|-----------|
| 0       | 0       | 0         |
| 0       | 1       | 0         |
| 1       | 0       | 0         |
| 1       | 1       | 1         |

- оператор ИЛИ (`||`) возвращает истину когда хотя бы один из операндов истинный;

| операнд | операнд | результат |
|---------|---------|-----------|
| 0       | 0       | 0         |
| 0       | 1       | 1         |
| 1       | 0       | 1         |
| 1       | 1       | 1         |

- оператор НЕ (`!`) возвращает истину когда операнд ложный;

| операнд | результат |
|---------|-----------|
| 0       | 1         |
| 1       | 0         |

Используя логические операторы в программе мы можем написать логику практически любой сложности. В языке С нет ограничений на использование сложных условий. Сложные условия это такие, где в круглых скобках выполняется более одного сравнения. Сравнения производятся в порядке заранее оговоренного приоритета. В списке ниже указаны операторы в порядке уменьшения их приоритета:

1. !
2. <, <=, >, >=
3. ==, !=
4. &&
5. ||

Приведём короткий пример: дана некоторая целочисленная переменная и нужно выяснить, не выходит ли эта переменная за рамки заданных значений, например от нуля до десяти. Условие можно будет скомбинировать так: `x >= 0 && x <= 10`. В случае его истинности - выдадим сообщение о том, что `x` подходит.

```
1     int x = 7;
2     if ((x >= 0) && (x <= 10)) {
3         printf("X Fits!\n");
4     }
5
```

В данной записи мы видим, что сначала `x` сравнивается с нулём, затем с десятию, и в конце результаты будут сравнены между собой.

```
$ ./program
X Fits!
$
```

Самый не приоритетный оператор тернарный, внимательный читатель мог заметить, что он даже не вошёл в список выше, это сделано, поскольку использование тернарного оператора внутри условий нежелательно. Тернарный оператор внутри условий резко снижает читаемость кода и усложняет его интерпретацию. Если Вы сомневаетесь в приоритете сравнений или Вам необходимо описать какое-то очень сложное условие, всегда можно воспользоваться простыми математическими круглыми скобками, задав приоритет операций явно. В таком случае в первую очередь будут выполнены операции в скобках.

### 5.3 Блоки кода и область видимости

Говоря об операторах языка C и управляющих конструкциях нельзя не сказать о «блоках кода» и «областях видимости». Как видно, условные операторы содержат записи в фигурных скобках. В такие же скобки заключён код функции `main`. Эти скобки

называются «операторными», а то что в них содержится, называется «блоком кода» или «телом» оператора или функции. Все переменные, которые инициализируются внутри блока кода, существуют и «видны» только внутри кодового блока. Поэтому пространство между операторными скобками также называют «областью видимости».

```
1  int x = 7;
2  if ((x >= 0) && (x <= 10)) {
3      int var = 0;
4      printf("X Fits!\n");
5  }
6  printf("%d", var);
7
```

На этом примере можно увидеть, что мы не можем напечатать значение переменной `var`, поскольку она была создана внутри блока кода оператора `if()` и перестала существовать для нашей программы как только мы вышли за его пределы. Такой код даже не скомпилируется:

```
$ gcc -o program main.c
error: 'var' undeclared (first use in this function);
   did you mean 'char'?
printf("%d", var);
           ^~~
           char
$
```

## 6 Циклы

Теперь, когда мы узнали, что можно делать в программе с использованием условий, пришло время познакомиться с таким базовым понятием программирования, как цикл.

Цикл - это одно или несколько действий повторяющихся до тех пор пока не наступит условие, прекращающее эти действия.

С помощью циклов в программировании выполняются все рутинные задачи, такие как поиск значений в больших наборах данных, создание разного рода прогрессий, построение графиков, сортировки, ожидание ответов на запросы, чтение потоков данных и многие другие.

### 6.1 Операторы циклов

#### 6.1.1 `while()`

Базовый цикл в языке C записывается при помощи ключевого слова `while()` после которого в круглых скобках пишется условие. При истинности данного условия будет выполняться тело цикла, которое в свою очередь пишется в фигурных скобках. Общий внешний вид очень похож на условный оператор, с той лишь разницей, что по окончании выполнения операторов внутри фигурных скобок мы переходим не на следующую строку, а обратно в проверку условия.

Для примера, выведем на экран все числа в заданном промежутке, границы которого мы обозначим как `a` и `b`. Для этого нам необходимо их инициализировать, то есть объявить и задать начальные значения, и пока `a` меньше `b` заходить в тело цикла где мы будем выводить на экран и инкрементировать меньшее число, пока оно не станет равным второму. Как только числа сравняются условие входа в тело цикла перестанет быть истинным, и мы не зайдём в него, оказавшись на следующей после тела цикла строке.

```
1  int a = 10;
2  int b = 20;
3  while (a < b) {
4      printf("%d ", a++);
5  }
```

Запустим программу и убедимся что все работает:

```
$ ./program
10 11 12 13 14 15 16 17 18 19
$
```

В данном коде мы использовали оператор инкремента в постфиксной записи, которая означает, что значение переменной `a` сначала будет передано функции вывода на экран, а уже потом увеличено на единицу.

Один проход тела цикла и возврат в управляющую конструкцию называется **итерацией**, этот термин можно очень часто услышать в беседах программистов, а зачастую и не только в беседах о программировании.

Также допустимо использовать префиксную запись оператора инкремента, при которой, как не сложно догадаться, значение сначала будет увеличено, а уже потом передано функции вывода на экран.

```
1   int a = 10;
2   int b = 20;
3   while (a < b) {
4       printf("%d ", ++a);
5   }
6
```

Запустим снова, чтобы увидеть разницу: после запуска первого варианта мы увидели в терминале значения от 10 до 19, а после запуска второго варианта значения от 11 до 20.

```
$ ./program
11 12 13 14 15 16 17 18 19 20
$
```

Давайте напишем ещё один пример, в котором подсчитаем сколько чётных чисел в промежутке от `a` до `b`. Для этого нам понадобится циклически пройти по всем числам от `a` до `b` и в каждой итерации цикла, то есть для каждого числа, сделать проверку, является ли число чётным. Если является, увеличить счётчик чётных чисел для заданного промежутка. Обратите внимание, что приращение значения переменной `a` происходит прямо в проверке условия. Обычно такой код считается трудночитаемым, поскольку сложно сразу обратить внимание на то, что в пятой строке происходит не только сравнение, но и пост-инкремент переменной `a`.

```

1  int a = 10;
2  int b = 20;
3  int evens = 0;
4  while (a < b) {
5      if (a++ % 2 == 0)
6          evens++;
7  }
8  printf("There are %d even numbers in sequence", evens);
9

```

После того, как перестанет выполняться условие в скобках, и наш цикл будет завершён, выведем в консоль получившееся количество чётных чисел. Запустим нашу программу и убедимся в правильности ее работы.

```

$ ./program
There are 5 even numbers in sequence
$

```

Программа выдала результат: пять чётных чисел, давайте пересчитаем вручную: 10, 12, 14, 16, 18, значение 20 не войдёт в результат работы программы, поскольку на последней итерации, когда  $a = 20$  условие в круглых скобках окажется ложным: двадцать не меньше двадцати. А раз условие не выполнено мы не попадаем в тело цикла.

Цикл `while()` используется когда мы не можем достоверно сказать сколько итераций нам понадобится выполнить. На этом примере мы видим что тело цикла может быть любой сложности, оно может содержать как условные операторы так и другие циклы.

### 6.1.2 `do {} while();`

Простой цикл `while()`, по очевидным причинам, попадает в категорию циклов с предусловием. Сначала программа проверяет условие, затем, по результатам этой проверки, либо выполняет либо не выполняет тело цикла. А раз есть циклы с предусловием, то логично предположить наличие циклов с постусловием. Как ни удивительно, в языке C есть и такой. Это разновидность цикла `while()`, который записывается как ключевое слово `do {тело цикла} while (условие);`, в этой конструкции сначала гарантированно один раз выполнится тело цикла, и только потом будет принято решение, нужно ли выполнять его снова. Такие циклы широко применяются для проверки пользовательского ввода до получения приемлемого результата и для ожидания ответов на запросы, что логично, поскольку глупо было бы ожидать ответ, не послав запрос.

Например, мы пишем калькулятор, и знаем, что в арифметике числа нельзя делить на ноль. Предположим, что наш пользователь этого не знает. Что ж, будем предлагать пользователю ввести делитель, пока он не введёт что-то отличное от нуля. Если пользовательский ввод равен нулю, значит нам нужно спросить его снова. Это и будет условием для очередной итерации цикла. Когда пользователь введёт удовлетворяющее нашему условию число произведём необходимые подсчёты и выведем их результаты в консоль.

```
1  int input;
2  do {
3      printf("Enter a divider for 100, ");
4      printf("remember, you can't divide by zero: ");
5      scanf("%d", &input);
6  } while (input == 0);
7
8  printf("100 / %d = %d", input, 100 / input);
9
```

Запустим программу. Каждый раз когда мы вводим ноль, программа будет повторно задавать нам вопрос.

```
$ ./program
Enter a divider for 100, remember, you can't divide by zero: 0
Enter a divider for 100, remember, you can't divide by zero: 0
Enter a divider for 100, remember, you can't divide by zero: 5
100 / 5 = 20
$
```

Когда введем любое другое число получим результат нашего деления.

### 6.1.3 for(;;)

Зачастую, складываются ситуации, когда мы точно знаем, сколько итераций цикла нам понадобится, например, когда мы последовательно проверяем содержимое созданного нами числового ряда, или заполняем значениями таблицы, границы которых заранее известны. В конце концов, для подсчёта среднего арифметического некоторого конечного числового ряда. В этих ситуациях принято использовать. Это цикл с условием, где заранее отведено место для инициализации переменной-счётчика, условия захода в следующую итерацию цикла и изменения переменной счетчика. В более поздних версиях C (C99) появилась возможность объявлять переменную счетчик прямо в управляющей конструкции.

Для применения более позднего стандарта в проекте, необходимо установить специальный ключ компиляции `-std=c99`, то есть полная команда компиляции исходного кода для транслятора GCC будет выглядеть так:

```
gcc program.c -std=c99 -o program
```

В классическом C необходимо объявить переменную счетчик заранее, а в управляющей конструкции можно только задать ей начальное значение. Условие захода в следующую итерацию цикла это логическое выражение, которое может принимать два значения: истина и ложь. Если условие истинно - идём на следующую итерацию (исполняем тело цикла), если ложно – выходим из цикла и перемещаемся на следующий после цикла оператор.

```
1 // classic style
2 int i;
3 for (i = 0; /*condition*/; /*increment*/) {
4     // body
5 }
6
7 // C99 and later
8 for(int i = 0; /*condition*/; /*increment*/) {
9     // body
10 }
11
12 // Example:
13 int i;
14 for (i = 0; i < 5; i++) {
15     printf("%d ", i);
16 }
17
```

Цикл из примера выше выведет на экран числа от нуля до четырёх. На каждой итерации мы будем инкрементировать значение `i`, соответственно, пока логическое условие `i < 5` верно, мы будем заходить в тело цикла, а как только `i` станет равным пяти, логическое условие вернет ложь (0) и мы из него выйдем.

```
$ ./program
0 1 2 3 4
$
```

## 6.2 Управление циклами

Операторы, которые осуществляют управление циклами называют операторами безусловного перехода, поскольку они просто перемещают выполнение программы на дру-

кую строку, невзирая ни на что. Программист должен сам описать логику такого перемещения, используя условные операторы.

**Оператор `continue`;** нужен для того, чтобы программа проигнорировала оставшиеся действия на текущей итерации цикла, часто используется для отбрасывания неподходящих значений при переборе больших объёмов данных. Оператор `continue` просто напросто передаёт управление логической конструкции цикла.

**Оператор `break`;** используется для того, чтобы выйти за пределы цикла, мы сразу попадаем к следующему после цикла оператору, без передачи управления логической конструкции.

### 6.3 Практические задачи

**Возведение в степень** Решим немного более сложную, чем выведение в консоль числовых рядов задачу возведения числа в степень. Язык C не предоставляет оператора возведения в степень по умолчанию, как это делают некоторые другие языки высокого уровня, такие как Python, поэтому для этой математической операции нам нужно подключать специальную математическую библиотеку. Но иногда это может оказаться излишним, ведь не так сложно написать собственную функцию, которая бы делала это.

Как известно, возведение в степень - это последовательное умножение основания на само себя указанное количество раз.

А раз заранее известно, сколько раз мы будем умножать основание само на себя, это работа для цикла `for ( ; ; )`. Объявим переменную-итератор `i`, переменную-основание, переменную-показатель и переменную, в которую будем сохранять промежуточные и конечный результаты.

```
1  int i;  
2  int base;  
3  int significative;  
4  int result = 1;  
5
```

Принцип возведения числа в степень следующий: результатом будет совокупность результатов предыдущих итераций умноженных на основание. А значит и алгоритм будет повторяться столько раз, сколько указано в показателе, на каждом повторении

умножая промежуточный результат на основание, сохраняя произведение обратно в переменную с промежуточным результатом.

```
1     for (i = 0; i < significantive; i++) {
2         result = result * base;
3     }
4
```

Запись вида `result = result * base;` можно сократить до `result *= base;`, и выведем результаты работы цикла в консоль.

```
1     for (i = 0; i < significantive; i++) {
2         result *= base;
3     }
4     printf("%d powered by %d is %d \n", base, significantive, result);
5
```

Конечно, мы можем спросить у пользователя какое число, и в какую степень он хочет возвести, для этого применим уже привычные нам конструкции. Так, весь код программы будет иметь следующий вид:

```
1 #include<stdio.h>
2
3 int main(int argc, char *argv[]) {
4     int i;
5     int base;
6     int significantive;
7     int result = 1;
8     printf("Enter base: ");
9     scanf("%d", &base);
10    printf("Enter significantive: ");
11    scanf("%d", &significantive);
12    for (i = 0; i < significantive; i++) {
13        result *= base;
14    }
15    printf("%d powered by %d is %d \n", base, significantive, result);
16 }
```

Запустим нашу программу, введем для базы значение два, для показателя десять. Убедимся, что наша программа работает корректно,  $2^{10} = 1024$ . Естественно, что существуют более оптимальные алгоритмы возведения в степень, например, возведение в степень с использованием свойства чётности степени,  $2^{10} = 4^5$ , то есть при чётном показателе можно возвести основание в квадрат, а показатель поделить на два, тем самым сократив количество итераций простого цикла, описанного выше, вдвое.

```
$ ./program
Enter base: 2
Enter significative: 10
2 powered by 10 is 1024
$
```

**Простое число** Решим ещё одну несложную задачу. Напишем программу, которая будет определять, является ли введённое пользователем число простым. Мы напишем не самый быстрый и оптимальный алгоритм, но постараемся использовать все доступные нам конструкции. То есть, эта задача призвана продемонстрировать возможности языка, а не улучшить или заменить существующие, более оптимальные алгоритмы проверки.

Что такое простое число? Это такое число, которое имеет ровно два делителя с целочисленным результатом - единицу и само себя.

Наша программа будет запрашивать у пользователя число и определять, простое оно или нет. Для этого заведём переменную, и привычными нам функциями, попросим пользователя ввести число, которое положим в неё. Для подсчетов нам понадобятся дополнительные переменные, например, переменная которая будет хранить количество делителей, назовем ее `dividers` и переменная итератор `i` значение которой будет увеличиваться от единицы до введенного пользователем числа.

```
1  int number;
2  int dividers = 0, i = 1;
3  printf("Enter number: ");
4  scanf("%d", &number);
5
```

Поскольку мы не знаем, сколько итераций понадобится, напишем цикл `while`, и пройдемся от единицы, которую мы записали в переменную `i` до введённого пользователем числа. После того как мы переберем все возможные варианты от единицы до введённого пользователем числа выведем в консоль получившийся результат. Напишем введённое число, а дальше предоставим программе выбор ставить ли частицу «не» при помощи заполнителя `%s` и тернарного оператора. В случае истинности условия `dividers == 2` тернарный оператор вернет пустую строку, в случае ложности, вернет частицу «не». Обратите внимание на то, как оператор вывода на экран написан в несколько строк. Такие разделения на несколько строк можно часто увидеть, если оператор длинный и может, например, выйти за пределы экрана.

```

1  while (i <= number) {
2      // here will be the check algorithm
3  }
4  printf("Number %d is%s prime",
5         number,
6         (dividers == 2) ? "" : " not"
7         );
8

```

Итак, что будет происходить на каждой итерации цикла? Мы будем проверять делится ли введённое пользователем число на текущее значение счётной переменной. Если остаток от деления исходного числа на текущий делитель равен нулю, то мы увеличим счетчик количества целочисленных делителей для данного числа.

Снова обратите внимание, что переменная-итератор, последовательно перебирающая делители, увеличивается прямо в круглых скобках оператора `if`. Таким образом автор хочет подчеркнуть, что в коде, который пишут живые люди, такие конструкции встречаются чаще, чем хотелось бы. Часто даже можно встретить инкремент счётчика прямо в проверке условия захода в тело цикла и это ещё сильнее влияет как на читаемость кода, так и на его понятность.

Тело цикла для нашей задачи примет следующий вид:

```

1  if (number % i++ == 0) {
2      dividers++;
3  } else {
4      continue;
5  }
6  if (dividers == 3)
7      break;

```

Если количество целочисленных делителей не изменилось, то мы прекратим текущую итерацию цикла при помощи ключевого слова `continue`. Как мы знаем, оператор `continue` передаст управление в логическую конструкцию цикла, заставив программу проигнорировать все дальнейшие инструкции в рамках текущей итерации. Если количество целочисленных делителей достигнет трёх, что будет означать нецелесообразность дальнейших вычислений, мы разорвем цикл при помощи ключевого слова `break`. Напомним, что операторы `break;` и `continue;` являются операторами безусловного перехода, а значит достигнув их программа передаст управление соответствующей строке безусловно и без предупреждений. И полный получившийся код приложения будет такой:

```

1 #include<stdio.h>
2
3 int main(int argc, char *argv[]) {
4     int number;
5     int dividers = 0, i = 1;
6     printf("Enter number: ");
7     scanf("%d", &number);
8     while (i <= number) {
9         if (number % i++ == 0) {
10            dividers++;
11        } else {
12            continue;
13        }
14        if (dividers == 3)
15            break;
16    }
17    printf("Number %d is%s prime",
18        number,
19        (dividers == 2) ? "" : " not"
20    );
21 }

```

Естественно, повторимся, этот код можно оптимизировать по множеству направлений, как минимум, сократив как количество проверок, так и границы проверок (нет смысла проверять числа больше, чем  $\sqrt{number}$ ). Дополнительно можно не проверять чётные числа, например.

```

$ ./program
Enter number: 2
Number 2 is not prime
$ ./program
Enter number: 7
Number 7 is prime
$ ./program
Enter number: 457
Number 457 is prime
$ ./program
Enter number: 1457
Number 1457 is not prime
$

```

## 6.4 Множественный выбор `switch () {}`

Пришло время поговорить об операторе множественного выбора `switch () {}`. Тем более, что теперь мы обладаем всеми необходимыми для этого знаниями. Оператор

множественного выбора используется когда мы хотим описать действия для какого-то ограниченного количества условий. В отличие от оператора `if()`, который может использоваться также и для проверки диапазонов значений.

Это не совсем точно, потому что `switch()` в языке C тоже может использоваться для проверки диапазонов значений, но это довольно редко применяется, а в C++ и других языках может вовсе не работать. Не стоит пугаться, увидев `case 5 ... 50`: это как раз проверка диапазона целочисленных значений от 5 до 50 включительно.

Удобство применения того или иного оператора, естественно, зависит от задачи. Довольно важным ограничением оператора `switch()` в языке C является то, что он умеет работать только с целыми числами. Для примера, напишем свой собственный маленький калькулятор. Сразу предположим, что калькулятором будет пользоваться внимательный человек, который понимает, что арифметические действия можно совершать только с числами, и умножать строки или символы - не получится.

Заведем в нашей программе переменные типа `float`, которые будут хранить операнды – числа над которыми будут производиться действия. И переменную типа `char` для хранения операции. Спросим у пользователя, какие числа он хочет посчитать, для этого используем уже привычную нам связку `printf()/scanf()`. Далее, таким же образом предложим пользователю ввести действие, действие мы закодируем в виде чисел:

- 1 – сложение;
- 2 – вычитание;
- 3 – умножение
- 4 – деление.

Здесь вступает в силу тот факт, что на ноль делить нельзя, поэтому нам нужно не дать пользователю ввести в качестве второго операнда "0", если в качестве оператора он выбрал деление. Конечно, это можно оставить на усмотрение пользователя, но мы, как сознательные программисты, не дадим пользователю в арифметическом порыве сломать нашу программу. Совершенно очевидно, что просто скопировать ввод будет неправильным действием. Для того чтобы не дать пользователю сделать неправильный ввод введем в программу условие: если пользователь выбрал деление, используя цикл `do {} while()`; будем просить его ввести второй операнд отличный от нуля, а если выбран-

ный оператор не является делением, то просто попросим пользователя ввести второе число, без проверок и повторений.

```
1  if (operator == 4) {
2      do {
3          printf("/nEnter second operand: ");
4          scanf("%f", &second);
5      } while (second == 0);
6  } else {
7      printf("/nEnter second operand: ");
8      scanf("%f", &second);
9  }
10
```

Если мы воспользуемся нашими знаниями на текущий момент, то мы напишем примерно следующее: если оператор равен единице, то делать это, в противном случае, если оператор равен двум, то делать вот это, и так далее, описывали бы все возможные действия условными операторами. Получился бы весьма громоздкий код.

```
1  if (operator == 1) {
2      //...
3  } else if (operator == 2) {
4      //...
5  }
6  //...
7
```

Но хорошая новость в том, что существует гораздо более удобный оператор `switch() {}`. Воспользуемся им относительно переменной `operator`. Разделим действия оператора `switch() {}` на несколько так называемых *кейсов*.

```
1  switch (operator) {
2      case 1:
3          result = first + second;
4      case 2:
5          result = first - second;
6      case 3:
7          result = first * second;
8      case 4:
9          result = first / second;
10     default:
11         printf("Unknown operator\n");
12     }
13
```

Оператор `switch() {}` последовательно проверит входящую переменную на со-

ответствие описанным в кейсах значениях. В случае, если значение совпадёт, будет выполнен блок кода до оператора `break;`, если же значение переменной не совпадёт ни с одним из описанный в кейсах, выполнится блок по умолчанию `default`.

Важно помнить, что в случае отсутствия внутри `case` оператора `break;`, программа будет выполнять последующие кейсы, пока не найдёт `break;` или пока не закончится конструкция `switch () {}`, то есть пока не встретится её закрывающая фигурная скобка.

В кейсах мы опишем присваивание результата в переменную `result`, а после выхода из `switch () {}` - вывод результата в консоль. Кейсом по умолчанию будет вывод пользователю сообщения о невозможности распознать оператор. Так получается, что даже если мы ввели неизвестный оператор, программа попытается вывести в консоль результат, что неприемлемо. Поэтому кейс по умолчанию должен содержать также и оператор `return 1;` вынуждающий программу экстренно завершиться с кодом ошибки 1. Запустив описанный нами калькулятор, убедимся что все работает. Сымитируем нерадивого пользователя и несколько раз попробуем ввести при использовании четвёртого оператора цифру ноль, программа естественно не даст нам этого сделать.

```
$ ./program
Enter first operand: 10
Enter 1 for (+), 2 for (-), 3 for (*), 4 for (/): 4
Enter second operand: 0
Enter second operand: 0
Enter second operand: 0
Enter second operand: 3
Result is: 3.333333
$ ./program
Enter first operand: 10
Enter 1 for (+), 2 for (-), 3 for (*), 4 for (/): 5
Unknown operator
$
```

Также приведём полный листинг получившейся программы в приложении А. Далее некоторые примеры будет невозможно привести полностью, поэтому собирать их в единый работающий код читатель будет вынужден самостоятельно.

## 7 Функции

### 7.1 Понятие функции, параметры и аргументы

Функция - это такая обособленная часть кода, которую можно выполнять любое количество раз. У функций обязательно в таком порядке должны быть описаны: тип возвращаемого значения, название, параметры и так называемое тело, то есть, собственно, исполняемый код. Рассмотрим более детально функцию `int main (int argc, char *argv[])`: указанный тип `int` - это *тип возвращаемого значения*, то есть на том месте, откуда будет вызвана эта функция, в результате её работы по факту выполнения оператора `return;`, появится некое целое число. Возвращаемые значения могут быть любых типов. В случае же когда функция не должна возвращать результат своей работы, или никакого возвращаемого результата не предполагается, указывается ключевое слово `void` (англ. - пустота). То есть на месте вызова функции, в результате её выполнения, не появится никакого значения (обычно, таким значением бывает `gvalue`). Оператор `return;` обязателен для не-`void` функций, а в `void` функциях может присутствовать или нет, но никогда не содержит возвращаемого значения. Написанное с маленькой буквы слово `main` - это *название функции*. Функция именно с таким названием, написанным с маленькой буквы, всегда является точкой входа в программу (2.2). Операционная система ищет именно эту функцию, когда получает команду на выполнение программы.

Названия функций в рамках одной программы не должны повторяться и не должны начинаться с цифр или спецсимволов, также, как и названия переменных (см стр. 6) никаких других ограничений на название функций не накладывается.

Конструкция в круглых скобках `(int argc, char *argv[])` - это *параметры функции*. Параметры функции - это такие переменные, которые создаются при вызове функции и существуют только внутри неё. С их помощью можно передать в функцию какие-то аргументы и исходные данные для работы. Параметры пишутся в круглых скобках сразу после названия функции. В случае если функция не принимает параметров необходимо поставить после названия пустые круглые скобки `( )`. Весь код, содержащийся в фигурных скобках после параметров функции называется *телом функции*. Это те операторы и команды, которые будут последовательно выполнены при вызове функции. В теле функции мы можем **вызывать** другие функции, но **никогда не можем объявлять, описывать или создавать в теле функции другие функции**. Никаких других ограни-

чений на написание тела функции язык не накладывает. Таким образом, общий вид функции следующий:

```
ТипВозвращаемогоЗначения Имя (СписокАргументов)
{
    ТелоФункции
    return ВозвращаемоеЗначение;
}
```

Далее приведём небольшой пример, который призван продемонстрировать, как выглядит простейшее *объявление* и *описание* функций (function declaration and definition), а также их вызов из функции `int main (int argc, char *argv[])`.

```
1 void somefunction() { // <-- this is a function
2     printf("some function\n");
3     // some useful things
4 }
5
6 int anotherFunction() {
7     printf("another function\n");
8     // more useful things happened
9     return 10;
10 }
11
12 int main (int argc, const char* argv[]) {
13     printf("main function\n");
14     // more useful things
15     somefunction(); // <-- this is invocation
16     int x = anotherFunction();
17     printf("x = %d\n", 10);
18     return 0;
19 }
20
```

Так, на шестнадцатой строке кода выше мы видим, что **вернувшееся** из функции, объявленной на шестой строке целое число 10 будет присвоено переменной `x` и выведено в терминал семнадцатой строкой.

```
$ ./program
main function
some function
another function
x = 10
$
```

Функции принято разделять на проверяющие, считающие и выводящие, и каждая из

вышеописанных функций не должна нести дополнительной нагрузки. То есть, функция не должна знать откуда в программе появились её параметры, и где будет использован результат её работы. То есть сам язык таких ограничений не накладывает, но такой подход к написанию функций делает их значительно более гибкими и даёт им возможность быть переиспользованными. Без применения такого подхода было бы невозможно писать абстрактные библиотеки и фреймворки.

**Параметры функции** - это те переменные, которые указываются в круглых скобках при определении или описании функции. Параметры функции существуют как локальные переменные в кодовом блоке тела функции. **Аргументы функции** - это те значения переменных или литералов, которые указываются в круглых скобках при вызове функции.

Для примера опишем функцию, суммирующую два числа. Для простоты, в качестве аргументов она будет принимать целые числа и возвращать целочисленный результат. Обратите внимание, что функция не «знает» откуда взялись эти числа, мы можем их прочесть из консоли, можем задать в виде констант или получить в результате работы какой-то другой функции. Внутри функции `int main (int argc, char *argv[])` программа вызывает нашу функцию `sum(int x, int y)` суммирующую два числа и передаём в качестве аргументов эти числа.

```
1  int sum(int x, int y) {
2      int result = x + y;
3      return result;
4  }
5
6  int main (int argc, const char* argv[]) {
7      int a;
8      scanf("%d", &a);
9      int x = sum(50, a);
10     printf("x = %d\n", 10);
11     return 0;
12 }
13
```

Обратите внимание, что в качестве аргументов мы можем передавать константные значения, а также переменные. Значения переменных мы можем получить например из консоли, либо в результате выполнения какой-нибудь другой функции.

Как уже было сказано, параметры - это переменные, которые хранят в себе некоторые начальные значения вызова функции. Параметризация позволяет использовать одни и те же функции с разными исходными данными. Приглядимся повнимательнее

```
$ ./program
x = 110
$
```

к хорошо знакомой нам функции `printf()`; . Строка, которую мы пишем в круглых скобках в двойных кавычках - это аргумент функции. То есть мы знаем, что функция умеет выводить на экран строки, как именно - нам нет дела, а какие именно строки - мы указываем в качестве аргумента. Функция `printf()`; примечательна еще и тем, что она может принимать в себя нефиксированное количество аргументов. Описание работы таких функций, а также их написание выходит далеко за пределы основ языка, нам важно помнить что мы можем это использовать. В аргументе функции `printf()` мы можем написать заполнитель соответствующего типа и, например, вызвать нашу функцию `sum()` .

## 7.2 Оформление функций. Понятие рефакторинга

Теперь мы без проблем можем оформить уже существующие у нас программы в виде функций. Например, оформим в виде функции программу проверки простоты числа. Для этого опишем функцию которая возвращает целое число, назовем ее `isPrime()` , в качестве параметра она будет принимать целое число, назовем его `number` . Найдем в предыдущих разделах (стр. 45) программу определения простоты числа и скопируем в тело функции. Внесем небольшие правки, уберем вывод так как это будет, можно сказать, классическая проверяющая функция, вывод оставим для функции `int main (int argc, char *argv[])` , пусть о наличии у нас терминала «знает» только она.

Такой процесс, перенос участков кода между функциями, выделение участков кода в функции, синтаксические, стилистические и другие улучшения, называется **рефакторингом**. Обычно, рефакторингом занимаются сами разработчики в свободное от основной деятельности времени, в периоды код ревью или по необходимости улучшить читаемость/повторяемость собственного кода.

Следовательно, допишем условия: если делителей два, то число простое, возвращаем ИСТИНУ, то есть любое ненулевое значение, в нашем примере - единицу. Если же делителей больше – число не простое, возвращаем ЛОЖЬ, в нашем случае, это ноль. Такой вывод можно записать и другим способом, `return (dividers == 2)` – это выражение в случае истины вернет единицу в случае лжи ноль. Или можно воспользоваться тернарным оператором, то есть, написать `return (dividers == 2) ? 1 : 0;`

если условие в скобках истинно вернется единица, ложно – ноль. Также важно, что выйти из функции мы можем на любом этапе ее выполнения, например если делителей уже три, то нам нужно не завершать цикл, а вернуть ЛОЖЬ из функции.

```

1
2 int isPrime(int number){
3     int dividers = 0, i = 1;
4
5     while(i <= number){
6         if(number % i++ ==0)
7             dividers++;
8         else
9             continue;
10
11         if (dividers == 3)
12             return 0;
13     }
14     return (dividers == 2)
15 }
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
26
```

значений и вывод на экран мы оставим в ней, а подсчёты, проверки или другие важные действия и алгоритмы положим в функции. Именно это абстрагирование является сильной стороной использования функций, так, например, у нас нет необходимости каждый раз вставлять в программу код взаимодействия с консолью при выводе каждой строки, а можно ограничиться вызовом функции `printf()` ;.

### 7.3 Прототип функции, заголовочные файлы

Зачастую возникают ситуации, когда функция не описана до точки входа в программу, или вовсе лежит в другом файле, возможно, даже написанном не нами. В этом случае мы должны сообщить компилятору, что такую функцию придётся дополнительно поискать. Для этого необходимо указать всю информацию о функции, кроме её тела. Такое объявление называется **прототип или определение функции** (англ. function definition).

С определением функции тесно связано понятие *сигнатуры* функции. Сигнатура функции для разных языков программирования представляется немного разным составом сведений, так, например, в языке C сигнатура - это тип возвращаемого значения, название функции и порядок типов параметров, например, для функции суммирования чисел, описанной выше, это будет `int sum(int, int)`.

Опишем прототип функции `isPrime()`, описав сигнатуру этой функции. Обратите внимание, что допустимо в определении функции также писать названия параметров, а не только их типы, но это необязательно.

```
1 int isPrime(int number);
```

Из таких определений часто составляют так называемые *заголовочные файлы*. Заголовочные файлы это мощный инструмент модульной разработки. Мы уже неоднократно видели подключение заголовочного файла `stdio.h`, Обнаружив данный файл на диске компьютера, мы увидим, что в нём содержатся другие подключения библиотек, директивы препроцессора (о которых более подробно мы будем говорить в следующих разделах) и прототипы функций (например, так часто используемой нами `printf()`). Заголовочным этот файл называется, потому что его обычно пишут в коде программы в самом верху, и, фактически, компилятор просто вставляет его содержимое в текст программы. Расширение файла (`.h`) является сокращением от английского слова *header*, заголовок. Обратите внимание, что подключая заголовочный файл `stdio.h` мы получаем вообще всю функциональность стандартного ввода-вывода, то есть, например,

работу с файлами, которую можем и не использовать. В стандарте C++20 было принято решение о переходе для поддержки повторяемости кода от заголовочных файлов к целостным модулям, импортируемым отдельно. Это позволяет интегрировать в программу только нужный функционал, игнорируя всю остальную библиотеку.

## 8 Указатели

Вот и пришла пора поговорить о серьёзном низкоуровневом программировании. О том, от чего стараются оградить программистов языки высокого уровня и современные фреймворки. Об указателях, что такое указатели и как они соотносятся с остальными переменными, что такое передача аргумента по значению и по указателю.. Этого разговора боятся все начинающие программисты и не без причин: работа с указателями на память может не только навредить программе, но и, например, оказать влияние на операционную систему (автор знает, что этот тезис не всегда справедлив, также, как тезис со стр. 9, но мы снова идём на такое упрощение ради того, чтобы было понятно, насколько это мощный инструмент). Также сразу отметим, что указателям достался свой собственный раздел в этом документе, хотя формально это просто ещё один тип данных.

Как мы, наверняка, помним, все переменные и константы, используемые в программе, хранятся в оперативной памяти. Оперативная память разделена на несколько участков, но не это для нас сейчас важно. Важно то, что у каждой переменной и константы в памяти есть свой собственный адрес. Адреса принято показывать на экране в виде шестнадцатиричных чисел. Этот адрес выдаётся нашей программе операционной системой, а язык C позволяет использовать его на усмотрение программиста. Иными словами в языке C есть возможность получить доступ к переменной не только по имени, но и по адресу. Получение доступа к значению переменной по адресу называется **разыменованием**. Давайте выведем в консоль всю имеющуюся информацию о переменной `a`. Мы знаем, что это целочисленная переменная значением 50, которая хранится по какому-то адресу.

```
1     int a = 50;
2     printf("value of 'a' is %d \n", a);
3     printf("address of 'a' is %p \n", &a);
4
```

Адрес переменной может храниться в специальной переменной, которая называется указатель. Для объявления указателя пишут тип переменной, адрес которой будет храниться в указателе, знак звёздочки и имя указателя. Такому указателю можно присвоить значение адреса существующей переменной, также как мы делали это раньше с другими типами данных. Для наглядности снова выведем всю имеющуюся у нас на данный момент информацию на экран. Напомню, для вывода адреса используется заполнитель `%p`. Выведем в консоль десятичное значение переменной `pointer` и адрес

переменной `pointer`. Увидим, что значение переменной `pointer` является как будто бы совершенно случайным числом, но ниже мы представим это значение не в виде обычного целого числа в десятичной системе счисления, а в виде адреса (заполнитель `%p`) то всё встанет на свои места.

```
1  int * pointer;
2  pointer = &a;
3
4  printf("value of 'pointer' is %d \n", pointer);
5  printf("address of 'pointer' is %p \n", &pointer);
6  printf("value of 'pointer' is %p \n", pointer);
7
8
```

Так, объединённый вывод двух предыдущих листингов будет примерно такой, и можно явно увидеть, что адрес `a` - это значение переменной `pointer`:

```
value of 'a' is 50
address of 'a' is 000000000061FE1C
value of 'pointer' is 6422044
address of 'pointer' is 000000000061FE10
value of 'pointer' is 000000000061FE1C
```

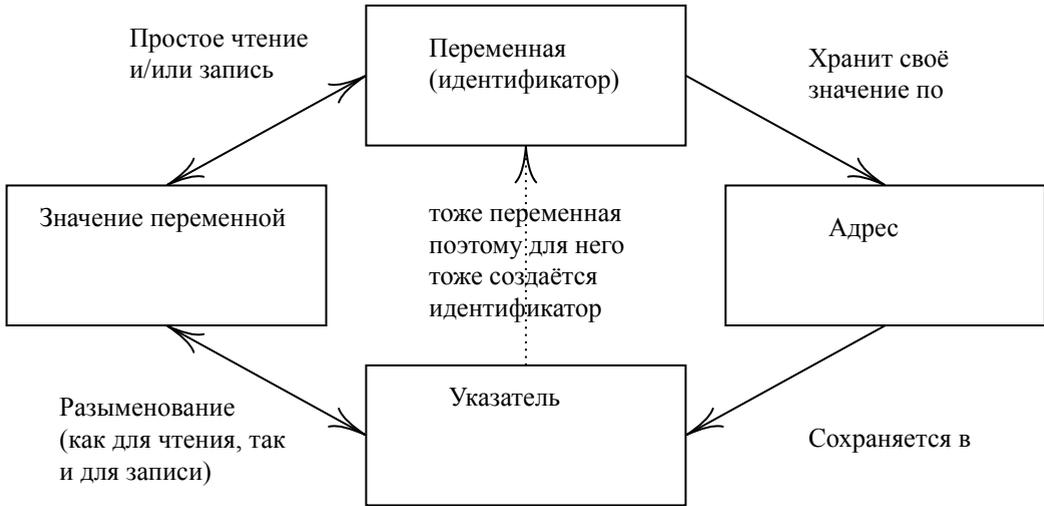
В общем-то, пока что ничего необычного, все эти операции мы выполняли на предыдущих уроках. Но поскольку `pointer` это немного необычная переменная, а указатель, то мы можем получить не только её значение, но и *значение переменной, на которую она указывает*, именно этот процесс называется разыменованием указателя. Давайте запишем, вывести в консоль «переменная `pointer` указывает на такое-то значение» и разыменуем `pointer`. То есть получим доступ к значению переменной, на которую ссылается указатель `pointer`.

```
1  printf("variable 'pointer' points at: %d", *pointer);
```

Таким образом, получается, что в указателе хранится ссылка на значение некоторой переменной, и мы можем это значение изменить. Давайте изменим значение переменной `a`, не на прямую, а с использованием указателя. Как видим, значение переменной изменилось.

```
1  *pointer = 70;
2  printf("value of a is %d \n", a);
```

То есть указатель - это простейший **ссылочный тип данных**. Без указателей невозможно себе представить создание классов, и всеми любимого объектно-ориентированного



2: Отношения указателей, адресов, идентификаторов и значений

программирования, даже массивов или строк. Теперь, когда мы знаем об указателях, и умеем получать значения переменных, на которые они указывают, а также изменять их, перед нами открываются невообразимые ранее перспективы. Мы можем писать функции не создавая в них копии переменных, а передавать в них указатели на уже существующие переменные, тем самым экономя память, и ускоряя выполнение программы. Например, не составит труда написать *программу*, которая бы меняла местами значения двух переменных. Но написать *функцию*, которая бы проделывала тоже самое невозможно без применения указателей. Почему? Очень просто - в параметре функции создаются свои собственные переменные, значения которых задаются копированием аргументов вызова, и меняются местами именно эти, скопированные значения в локальных переменных. И даже если мы вернём одно из этих значений – как быть со вторым? А получить доступ к значению второй переменной мы не можем, поскольку, помним, она находится в области видимости функции и недоступна извне. Такая передача аргументов называется *передачей по значению* (мы берём значение некоторой переменной и копируем внутрь функции, иногда такую передачу значений ещё называют *передачей копированием*). Т.е. мы берем значения некоторых переменных в функции `int main (int argc, char *argv[])` и передаем их в функцию, где создаём новые переменные с этими, переданными, значениями.

Как решить эту проблему? Передавать не значения переменных, а их адрес, тем самым сообщив функции, что нужно не создавать новые копии переменных, а сделать

что-то с уже существующими, и, естественно указать адрес, с какими именно. Передача в качестве аргумента адреса, и создание в теле функции нового указателя называется *передачей по указателю*.

Для языка C также справедливо выражение «передача по ссылке», поскольку в языке C нет отдельной операции передачи по ссылке. Так, например, в языке C++ передача по ссылке и передача по указателю - это разные операции.

То есть функция будет ссылаться на переменные, на которые мы укажем и оперировать их значениями. Давайте немного модифицируем нашу программу обмена значениями внутри двух переменных (4.3): опишем её в виде функции, принимающей в качестве параметров два указателя на целые числа типа `char`, и передадим адреса созданных в `int main (int argc, char *argv[])` переменных. Внутри функции, при её вызове, у нас будут создаваться не переменные, а указатели на переменные, то есть мы будем ссылаться на те самые переменные, созданные вне функции, и будем менять именно их (тех переменных) значения. Таким образом, нам не нужно ничего возвращать, потому что в функции ничего не создавалось, и типом возвращаемого значения функции должен быть `void`.

```
1 #include <stdio.h>
2
3 void swap(char* a, char* b) {
4     *a ^= *b;
5     *b ^= *a;
6     *a ^= *b;
7 }
8
9 int main(int argc, char* argv[]) {
10     char a = 11;
11     char b = 15;
12     printf("a = %d, b = %d", a, b);
13     swap(&a, &b);
14     printf("a = %d, b = %d", a, b);
15     return 0;
16 }
```

```
1 #include <stdio.h>
2
3
4
5
6
7 int main(int argc, char* argv[]) {
8     char a = 11;
9     char b = 15;
10    printf("a = %d, b = %d", a, b);
11    *a ^= *b;
12    *b ^= *a;
13    *a ^= *b;
14    printf("a = %d, b = %d", a, b);
15    return 0;
16 }
```

Применение такого подхода открывает перед нами широкие возможности. Важно, на схеме со стр. 58, что указатель - это тоже переменная, поэтому мы можем создавать указатели на указатели, и так далее, указатели любой сложности, тем самым увеличивая уровень абстракции программы.

## 9 Массивы

В этом разделе нас с вами ждут массивы. Много массивов. И ещё пара слов о директивах компилятору, иногда также называемых директивами препроцессора. С них и начнём.

### 9.1 Директива `#define`

Помимо уже хорошо знакомой вам директивы `#include`, частично описанной в разделе 2.2, естественно, существуют и другие. Некоторые из них ограничивают импорт описанных в заголовочном файле функций, некоторые «описывают» какие-то константы и даже действия. Вот, директиву **описать** мы и рассмотрим подробнее. Она не зря называется директивой препроцессора, поскольку даёт указание не процессору во время выполнения программы выделить память, присвоить значения, а непосредственно компилятору: заменить в тексте программы одни слова на другие. Таким образом можно задавать константы проекта, и даже делать сокращённые записи целых действий. Например, написав `#define ARRAY_LENGTH 50` мы предпишем компилятору, перед запуском трансляции нашего кода заменить все слова `ARRAY_LENGTH` на цифру 50. В такой записи, слово `ARRAY_LENGTH` будет называться *макроконстантой*.

Обратите внимание, что директива пишется немного не так, как обычный оператор языка, хоть и может находиться в любом месте кода. В конце директивы не ставится точка с запятой. Это важно именно потому что директивы работают с текстом программы, то есть если точка с запятой всё же будет поставлена, текст программы будет всегда содержать вместо макроконстанты число и точку с запятой, что может в корне изменить смысл программы.

Весьма удобно, но этим можно не ограничиваться, мы можем попросить компилятор заменить вызовы функций и операторы на короткие, удобные нам слова. Важно помнить, что директивы препроцессора работают с текстом программы, поэтому не осуществляют никаких дополнительных проверок. Это сложный и мощный инструмент, который чаще всего используется для решения нетривиальных задач, например, выбор кода, который попадёт в компиляцию в зависимости от операционной системы. Иногда в программах можно встретить описание недостающего, но такого привычного булева типа при помощи директив препроцессора:

```
1 #define bool int
2 #define true 1
3 #define false 0
```

Но нам пока что достаточно умения создать глобальную именованную константу. Код ниже демонстрирует, что директивы не обязательно группировать именно в начале файла, а можно использовать там, где это удобно и уместно, так мы можем объявить константу с длиной массива в начале файла, а можем прямо внутри функции `int main (int argc, char *argv[])`.

```
1 int main(int argc, char* argv[]) {
2     #define ARRAY_LENGTH 50
3     int a = ARRAY_LENGTH;
4     printf("a = %d", a);
5     return 0;
6 }
```

Результатом работы этой функции будет ожидаемое:

```
$ ./program
a = 50
```

## 9.2 Массивы

Вступление про директивы препроцессора напрямую не связано с темой массивов, но директива `#define` для объявления размера массива применяется чрезвычайно часто. Рассмотрим природу этого явления чуть позже.

Массив – это множество данных одного типа, расположенных в памяти подряд.

Язык C строго типизирован, поэтому невозможно создать массив из разных типов данных. На данном этапе мы рассматриваем только простые типы данных, поэтому и массивы будем рассматривать статические. Статическим массивом называют массив, количество элементов которого заранее известно и не изменяется за время работы программы. Альтернативой статическому массиву является динамический, таких массивов в языке C не существует, но всегда можно самостоятельно описать такую структуру данных, которая будет хранить значения, динамически расширяясь и сужаясь. Также для начала ограничим нашу беседу одномерными массивами, то есть такими, которые можно записать в виде значений через запятую. Статические одномерные массивы принято объявлять двумя способами:

- простое объявление с указанием размера;
- объявление, совмещённое с инициализацией

Для примера объявим массив, содержащий элементы типа `int`, дадим ему идентификатор или имя массива `arr` (сокращённо от англ `array`), укажем максимальное количество элементов которые может вместить в себя массив, например, пять. Как уже говорилось

```
1  int arr[5];
2
3  arr[0] = 10;
4  arr[1] = 20;
5  arr[2] = 30;
6
```

массив это множество данных или элементов. К каждому элементу массива можно обратиться по его номеру, который принято называть индексом. Индексация элементов начинается с нуля. Давайте заполним наш массив значениями типа `int`. Для этого последовательно обратимся к каждому элементу и присвоим значение. Обратите внимание, что язык C не гарантирует что инициализационное значение элементов массива будет равно нулю, если это не указано явно, поэтому выведя на экран содержимое массива, мы можем гарантировать значения только первых трёх элементов, которые мы указали в коде. Второй способ объявления, совмещённый с инициализацией массива используют, если массив сравнительно небольшой и его значения заранее известны, например:

```
1  int arr[6] = {1, 1, 2, 3, 5, 8};
```

При этом, если сразу заполняются все элементы, размерность можно не указывать. Итак, мы научились создавать и заполнять значениями массивы. Теперь общее правило объявления массивов в C: при объявлении массива нужно указать его имя, тип элементов, количество элементов, опционально - указать сами эти элементы. Количество элементов есть натуральное число, то есть целое положительное, ноль не может быть количеством элементов. Нельзя задавать переменное количество элементов массива. Так мы обязаны создавать массивы только с точно указанным числом элементов.

```
1  int nArr[100];    // An array for 100 int's;
2  float fArr[5];   // An array for 5 float's;
3  char cArr[2];    // An array for 2 char's;
4  int varElem;
5  int nArr[varElem]; // Compile error! Number of elements must be constant;
6
```

Для языка C это позволено сделать объявлением константы времени исполнения `const int elements; int arr[elements]`, но, например, в C++ такая запись вызовет

ошибку компиляции, поэтому там необходимо строго указывать размер числовым литералом или объявив его директивой `#define`, что, фактически, одно и то же.

В более поздних стандартах C++ появилось ключевое слово `constexpr`, позволяющее объявлять константы времени компиляции и отказаться от объявления размеров массива только литералом

Теперь давайте научимся получать доступ к элементам массива. Нет ничего проще, тем более, что мы это уже делали объявляли массив и для примера его заполняли. Для доступа к конкретному элементу массива нужно указать имя массива и индекс элемента в квадратных скобках. Квадратные скобки - это тоже оператор языка, он называется оператором индексного доступа: При помощи массивов решают множество задач, та-

```
1     int a = arr[0];
2     printf("lets see whats in 0-th element: %d", a);
3
```

ких как поиск, сортировка, составление таблиц соответствия, создание частотных диаграмм. На основе массивов создают более сложные структуры данных. Для короткого минимального примера, давайте напишем программу, которая будет печатать наш массив в консоль. Такая несложная программа даст нам следующий результат (Обратите

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4     int arr[5];
5     int i;
6     printf("Your array is: ");
7     for (i = 0; i < 5; i++) {
8         printf("%d ", arr[i]);
9     }
10    return 0;
11 }
```

внимание, что при такой инициализации, а точнее её отсутствии, значения внутри массива не гарантируются. В результате запуска программы на компьютере автора первые четыре индекса оказались равными нулю, а пятый принял странное отрицательное целочисленное значение):

```
$ ./program
Your array is 0 0 0 0 -497067408
```

Мы научились создавать, инициализировать массивы и обращаться к его элементам. Теперь решим задачу посложнее: напишем программу, которая проверит насколько статистически хорош описанный в стандартной библиотеке (языка C) генератор псевдослучайных чисел (функция `rand()` ;). Для такой статистической проверки нам понадобится сформировать так называемый *частотный массив*, массив, в котором будет содержаться информация о том, сколько раз то или иное число появилось во множестве значений, полученном при помощи генератора псевдослучайных чисел, частота вхождения значений. Сама генерация псевдослучайных чисел происходит при помощи функции `rand()` ; которая создаёт целое число типа `int`. Но, поскольку целое число в таком диапазоне нам не нужно, мы его сократим при помощи оператора получения остатка от деления. Обратите внимание на 14ю строку: для сгенерированного на

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #define ARRAY_LENGTH 10
5 #define NUMBERS_AMOUNT 1000000
6
7 int main( int argc, char *argv[]){
8     srand(time(NULL)); // initialize PRNG
9     int frequency[ARRAY_LENGTH] = {0};
10    int a;
11    int i;
12    for (i = 0; i < NUMBERS_AMOUNT; i++) {
13        a = rand() % ARRAY_LENGTH;
14        frequency[a]++;
15    }
16
17    for (i = 0; i < ARRAY_LENGTH; i++) {
18        printf("Number %d generated %6d (%5.2f%%) times\n",
19            i,
20            frequency[i],
21            ((float)frequency[i] / NUMBERS_AMOUNT * 100));
22    }
23    return 0;
24 }
```

13й строке числа 0 увеличим значение в 0-й ячейке массива, для числа 1 - в 1-й, и т.д. Данная программа наглядно демонстрирует не только работу с массивами, но и то, что генератор псевдослучайных чисел в языке C генерирует статистически верную последовательность случайных чисел. Инициализация генератора псевдослучайных чисел на восьмой строке `srand(time(NULL))` ; происходит значением текущего времени системы, то есть мы гарантируем, что начальное значение генератора будет отличаться от запуска к запуску, а значит наше исследование будет чуть более достоверным.

Запуск программы (даже несколько запусков) показывает, что всех значений полу-

```

$ ./program
Number 0 generated 99955 (10.00%) times
Number 1 generated 99977 (10.00%) times
Number 2 generated 100156 (10.02%) times
Number 3 generated 99952 (10.00%) times
Number 4 generated 100212 (10.02%) times
Number 5 generated 100713 (10.07%) times
Number 6 generated 99418 ( 9.94%) times
Number 7 generated 99768 ( 9.98%) times
Number 8 generated 99918 ( 9.99%) times
Number 9 generated 99931 ( 9.99%) times

```

чилось около десяти процентов, что говорит о том, что последовательность псевдослучайных чисел статистически верна.

### 9.3 Идентификатор массива

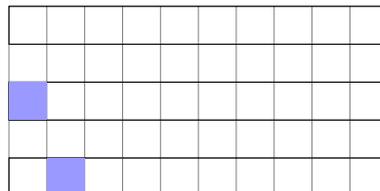
Это будет непросто, но мы поговорим о том, что из себя представляет идентификатор массива, чем чреват выход за пределы массива, затронем тему арифметики указателей и научимся передавать указатели на массивы в функции. Как упоминалось ранее, массив - это ссылочный тип данных. То есть в идентификаторе хранится адрес, ссылка на первый байт первого элемента массива, дальнейший доступ к элементам осуществляется посредством *смещения относительно этого байта*. Таким образом запись вида `array[0]` говорит нам о том, что нужно взять адрес массива и сместить указатель на 0 элементов того типа, из которых состоит массив. Отсюда становится ясно, почему **индексирование массивов начинается с нуля**.

Давайте попробуем визуализировать положение вещей в массивах. На рисунке со стр. 65 можно увидеть создание массива на первой строке, далее доступ к двум его ячейкам для записи в них значений. Соответственно, когда мы берем элемент с нуле-

```

1  int arr[ARRAY_LENGTH];
2  arr[0] = 20;
3  arr[1] = 50;
4

```



3: Визуализация создания и индексации массива

вым смещением, то есть по нулевому индексу, это будет ячейка памяти, находящаяся

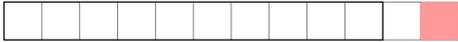
ровно по адресу массива. Все остальные ячейки мы просто игнорируем. Далее, берем следующий элемент массива. Для этого возьмем смещение в единицу, допустим, это будет число 50. Соответственно смещение будет на одну ячейку типа `int`. То есть мы игнорируем нулевой элемент и берем первый. Становится очевидно, что если мы в своём коде напишем такой индекс, который находится за пределами описанного массива - мы просто получим какое-то значение, которое никак не можем прогнозировать.

Относительно выхода за пределы массива надо сказать, что ни компилятор, ни тем более операционная система никаких проверок не делают, поэтому такие проверки (чаще всего самопроверки на этапе написания кода) полностью ложатся на плечи программиста. Язык C не предоставляет никаких сред виртуализации, никаких подсистем исключений, только выдача случайных данных, которые могут попасться нашей программе. Или запись в ячейки, которые совершенно не гарантированно останутся пустыми.

В связи с тем, индекс массива - это значение смещения, относительно его начала, и, как следствие, индексы массива всегда отсчитываются с нуля, важно помнить, что при создании массива из, например, десяти элементов десятый индекс будет находиться за пределами массива.

Надо сказать, что всё-таки бóльшая часть значений за пределами массива будет равна нулю, но всё равно лишний раз экспериментировать не стоит. Как мы уже знаем, в

```
1   arr[11] = 60;
2
```



#### 4: Выход за пределы массива

идентификаторе массива хранится ссылка на первый байт первого элемента массива, т.е. идентификатор является, по сути, указателем. Но существует несколько отличий: указатель - это переменная, к ней применимы, например, операции инкремента и декремента, чего конечно нельзя делать с идентификатором массива.

Идентификатор массива **не является** lvalue, Но один элемент массива является lvalue. Так, записи вида `arr++;` или `arr = 5` будут являться ошибочными, поскольку в них происходит обращение не к конкретному элементу с целью его изменения, а к массиву целиком.

Обратившись к идентификатору массива мы можем получить доступ к элементам массива не только при помощи записи индекса в квадратных скобках, но и при помощи так называемой арифметики указателей. Мы знаем, что массив - это единая область памя-

ти, и значения в нём располагаются подряд по очереди, значит, отсчитав от указателя на первый индекс нужное количество байт - мы получим указатель на второй индекс. Давайте для примера подсчитаем среднее арифметическое всех чисел в массиве, с использованием арифметики указателей.

Будем запрашивать значения для расчётов у пользователя. Создадим вспомогательную переменную `float result;`, для хранения результата и в цикле будем запрашивать у пользователя числа. Количество введенных цифр должно соответствовать количеству элементов массива, поэтому условием выхода из цикла будет равенство счётчика и последнего индекса массива, то есть, длины массива минус единица.

Обратите внимание, что в коде мы указали условием выхода из цикла **строгое неравенство**, то есть со значением `i` равным длине массива в тело цикла мы не попадём.

Помним, что индексация массива начинается с нуля, поэтому длина массива всегда на единицу больше последнего индекса. Выведем в консоль надпись «введите значение», при помощи функции `scanf()`; считаем его и сразу привычным образом, оператором индексного доступа, положим в массив. Выведем в консоль получившийся массив

```
1 int arr[ARRAY_LENGTH];
2 int i = 0;
3 float result = 0;
4 while (i < ARRAY_LENGTH) {
5     printf("Enter value %d: ", i);
6     scanf("%d", arr[i]);
7     i++;
8 }
9
```

при помощи цикла `for(;;)` и привычной нам функции `printf()`; . Следом напишем ещё один цикл в котором подсчитаем среднее арифметическое. Для этого к результату будем прибавлять существующий результат и значение массива на которое указывает уже не такая привычная, как квадратные скобки конструкция `*(arr + i)`, которую сразу и разберём. Как вы видите, некоторые подсчеты программа выполняет за нас - мы прибавляем к указателю единицу, двойку, тройку и т.д, а программа понимает, что надо взять не следующий по счёту байт, а следующий указатель. Так как в данном примере мы используем массив в котором хранятся значения типа `int`, а как вы помните `int` в подавляющем большинстве случаев - это четыре байта, то при увеличении указателя на единицу, мы обратимся к области памяти находящейся на четыре байта дальше идентификатора, при увеличении на двойку на восемь байт и так далее. Подсчитать среднее

```

1 printf("Your array is: ");
2 for (i = 0; i < ARRAY_LENGTH; i++)
3     printf("%d ", arr[i]);
4
5 printf("\nAnd the average is: ");
6 for (i = 0; i < ARRAY_LENGTH; i++)
7     result += *(arr + i);
8 printf("%f\n", result / ARRAY_LENGTH);
9

```

арифметическое не составит труда, этот алгоритм нам знаком со средней школы. Далее при помощи функции `printf()`; выведем в консоль среднее арифметическое. Запустим, повводим цифры и убедимся что все работает.

```

$ ./program
Enter value 0: 1
Enter value 1: 2
Enter value 2: 3
Enter value 3: 4
Enter value 4: 5
Enter value 5: 6
Enter value 6: 7
Enter value 7: 8
Enter value 8: 9
Enter value 9: 10
Your array is: 1 2 3 4 5 6 7 8 9 10
And the average is: 5.500000

```

Внимательный читатель мог заметить, что мы применяем операцию *разыменования*. Что происходит, когда мы таким образом обращаемся к массиву? Операция разыменования получает доступ к значению, находящемуся по адресу. Адрес массива - это адрес его первого элемента, поэтому конструкция `*arr` вернёт значение нулевого элемента массива. А прибавление значений к этому указателю будет смещать его также, как это делает оператор квадратных скобок.

Как уже упоминалось, идентификатор массива - это не обычный указатель. Обычный указатель хранит в себе адрес какой-то другой переменной, и сам где-то хранится. Указатель на начало массива хранит в себе адрес массива, то есть адрес его нулевого элемента, и сам этот указатель находится в этом самом месте. На первый взгляд сложновато? Но пусть Вас это не сбивает с толку, на деле всё не так жутко. На деле это означает, что при передаче массива (читай идентификатора массива) в функцию в качестве

аргумента, мы не должны использовать оператор взятия адреса, поскольку идентификатор массива сам по себе является указателем на собственное начало. Это открывает для нас широкие возможности по написанию функций, работающих с массивами данных. В только что написанной нами программе оформим вывод массива на экран и поиск среднего арифметического в виде функции. Опишем функции `printArray()` и `average()` в которые передадим указатель на массив и его длину, т.к. в массиве не содержится сведений о его размере.

Поскольку мы передаём в функцию указатель, то все действия которые описаны в этой функции будут происходить с массивом который мы создали в основной части программы через этот указатель, который мы передали, никакого копирования значений или чего то подобного. Для корректной работы наших функций объявим в них счётчик и изменим названия переменных на названия параметров.

```
1 void printArray(int* array, int length) {
2     int i;
3     for (i = 0; i < length; i++)
4         printf("%d ", array[i]);
5 }
6
7 float average(int* array, int length) {
8     float result = 0;
9     int i;
10    for (i = 0; i < length; i++)
11        result += *(array + i);
12    return result / length;
13 }
14
```

Так, полный листинг этого примера в приложении В

## 9.4 Многомерные массивы

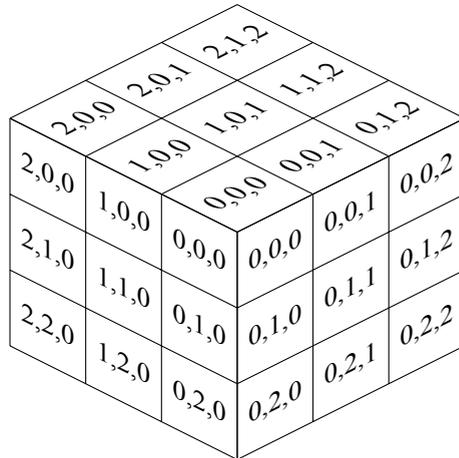
Массив в языке C может иметь сколько угодно измерений. Все массивы, с которыми мы имели дело до этого момента - одномерные, их легко визуализировать в виде простого перечисления элементов, возможно, как строки или как таблицы, состоящей из одной строки. Самые распространённые многомерные массивы - это двумерные и трёхмерные, которые легко себе представить в виде таблицы или куба соответственно. Итак, массив это структура, содержащая элементы. Двумерный массив - это массив из массивов, содержащих элементы. Трёхмерный - это массив из массивов, содержащих массивы, которые содержат элементы. И так далее. В массиве могут находиться любые типы данных, мы, для удобства, будем рассматривать работу массивов с числами.

Попробуем визуализировать двумерный массив. Создадим двумерный массив в коде, например, 5x5 элементов. Массив 5x5 – это 5 столбцов и 5 строчек. Соответственно, *каждая строчка – это будет у нас младший индекс, а каждый столбец – старший индекс*. Трёхмерный массив может быть, например, 3x3x3 – его можно визуализировать как всем известный кубик Рубика то есть, это три стоящих друг за другом таблицы 3x3. Также опишем его в коде ниже. Получается, что мы к таблице (ширине и высоте) добавили третье **измерение**, поэтому и массив получается **многомерным**, в данном случае, **трёхмерным**. Массивы бóльших размерностей тоже можно встретить в программах, но значительно реже, только лишь потому, что их действительно немного сложнее представить себе.

```
| int twoDimensional[5][5];
```

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| 0,0 | 0,1 | 0,2 | 0,3 | 0,4 |
| 1,0 | 1,1 | 1,2 | 1,3 | 1,4 |
| 2,0 | 2,1 | 2,2 | 2,3 | 2,4 |
| 3,0 | 3,1 | 3,2 | 3,3 | 3,4 |
| 4,0 | 4,1 | 4,2 | 4,3 | 4,4 |

```
| int threeDimensional[3][3][3];
```



Как работать с многомерными массивами мы рассмотрим на примере двумерного массива. Поставим для себя задачу: сформировать таблицу Пифагора (раньше такие на тетрадах в клетку печатали на обратной стороне). Таблица подразумевает наличие двух измерений: строк и колонок. Для этого объявим константы `rows` и `cols` и присвоим им значения 10, `rows` – это количество строк, а `cols`, соответственно, столбцов. Создадим двумерный массив, `table[rows][cols]`. Таким образом, мы создали массив размером `rows`, в каждом элементе которого содержится ссылка на массив размером `cols`, т.е. массив массивов содержащих непосредственные значения.

Важно помнить, что описанное строение массива является понятийным, и физически в памяти элементы хранятся иначе, но на основе этого понятия можно изучить работу оператора разыменования указателей в части работы с многомерными массивами, так для получения доступа к значению с индексом `cube[1][1][1]` нужно трижды разыменовать `cube`:

```
* ( * ( * (cube + 1) + 1) + 1)
```

Таблица Пифагора представляет собой таблицу, где строки и столбцы озаглавлены множителями, а в ячейках таблицы находится их произведение. Вот это самое произведение мы и будем сначала считать и записывать в массив, а затем выводить на экран. Заполнение многомерных массивов значениями ничем не отличается от заполнения одномерных массивов. Заполним нашу матрицу поэлементно: напишем двойной цикл который будет заполнять таблицу. Объявим переменные-итераторы. И с помощью внешнего цикла `for(;;)` пройдемся по всем строкам массива, а с помощью вложенного – по всем столбцам каждой строки массива, при этом будем записывать в каждый элемент (ячейку таблицы) результат умножения. Формула  $(r + 1) * (c + 1)$  позволяет исключить нули из нашей таблицы умножения. И ещё раз: внешний цикл при каж-

```
1 const int rows = 10, cols = 10;
2 int table[rows][cols];
3
4 int r, c;
5 for (r = 0; r < rows; r++) {
6     for (c = 0; c < cols; c++)
7         table[r][c] = (r + 1) * (c + 1);
8 }
```

дой итерации перемещает нас на одну строчку вниз. Вложенный, при каждой итерации, перемещает нас на одно значение вправо. Важно понять, что на одну итерацию внешнего цикла приходится `cols` итераций вложенного. Т.е. с помощью такой конструкции мы всегда поочередно переберём все элементы массива.

Сразу отметим, что помимо такого, поэлементного способа, также есть способ заполнять многомерные массивы посредством конструкции в фигурных скобках, этот подход часто используется когда мы заранее знаем все значения и их не очень много. Например, вот так объявляем и инициализируем двумерный массив размером 3x4:

```
1 int arr[3][4] = {
2     {0, 1, 2, 3},
3     {4, 5, 6, 7},
4     {8, 9, 10, 11}
5 };
```

Как и в случае такой инициализации одномерного массива, указание размерности в квадратных скобках не обязательно. Обратите внимание, что при такой инициализации удобно писать каждую строку массива отдельной строкой в коде, соответствующим образом расставляя фигурные скобки и запятые, такое оформление кода значительно улучшает его читаемость. Вернёмся к таблице Пифагора и напишем такой же (как и для заполнения) двойной цикл, который будет выводить на экран наши двумерные массивы в удобном для нас виде.

```
1 for (r = 0; r < rows; r++) {
2     for (c = 0; c < cols; c++)
3         printf("%3d ", table[r][c]);
4     printf("\n");
5 }
```

Запустив нашу программу с формированием таблицы умножения увидим, что все отлично работает. Полный листинг программы приводить не целесообразно, поскольку это цикл с заполнением и цикл с выводом, полностью приведённые выше. К тому же, такой код носит исключительно академический характер, и в случае действительной необходимости формирования таблицы Пифагора на экране промежуточное заполнение массива будет излишним, результат умножения целесообразнее сразу выводить на экран.

Как уже говорилось, все массивы могут содержать данные любых типов, в том числе и указатели. Именно это позволяет массиву хранить другие массивы, строки и прочие ссылочные типы данных. Используя массивы указателей, мы можем создать, например, массив строк.

```
1 char* stringArray[3] = {"Hello", "C", "World"};
2 int r;
3 for (r = 0; r < 3; r++)
4     printf("%s ", stringArray[r]);
```

Это указатели на строки, а точнее, на строковые литералы. Такой тип данных (строковый литерал) является указателем. И мы можем создать из этих указателей массив. Используя массивы указателей, мы можем создать, например, двумерный массив, где каждый элемент не обязан быть того же размера, что и остальные (но обязан быть того же типа, как мы помним). Но строки и сложно составленные указатели - это темы, которые очень сильно выходят за рамки Основ языка, хотя, конечно, это не мешает нам немного подробнее разобраться со строками в следующем разделе.

## 10 Строки

### 10.1 Основные понятия

Получив в предыдущих разделах представление об указателях и массивах, и вскользь несколько раз упомянув строки, пришла пора изучить их подробнее.

Итак, что же такое **строка**. В повседневной жизни строка *это набор или последовательность символов*. Так вот в языке С строка - это тоже последовательность символов. А последовательности значений, в том числе символьных, в языке С представлены, как вы, уже знаете, массивами и указателями. Никакого примитива `string` в языке С нет. Как бы нам этого не хотелось - его нет. Но есть и хорошая новость, примитива `string` не существует и в других языках (здесь имеются ввиду, конечно, си-подобные языки, то есть примерно треть вообще всех языков программирования, известных сегодня). Раз строка - это массив или указатель, это всегда ссылочный тип данных. Например, строку можно объявить двумя способами:

```
1 // строка
2 char str[50] =
3     {'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '\0'};
4
5 // литерал
6 char* str = "Hello world";
7
```

Раз строка - это набор символов, давайте немного вспомним что такое символы и как с ними работать. Как вам уже известно, символьная переменная это переменная типа `char`. В отличие от строки это примитивный, числовой, тип данных, и к нему применимы все операции допустимые для примитивов, такие как присваивание, сложение, вычитание, умножение, деление, хотя не все имеют смысл, так автор, например с трудом может представить ситуацию в которой необходимо умножать или делить коды символов, кроме, пожалуй, криптографии. Обратим внимание на следующую запись: Здесь

```
1 char c0 = 75;
2 char c1 = 'K';
3
```

значением `c0` является `75`, что абсолютно эквивалентно значению переменной `c1`, равной символу `K`. То есть можно сказать, что преобразование чисел в символы и наоборот согласно таблицы, наподобие ASCII, частично приведённой на стр. 75, встроена в рабо-

ту компилятора языка С. Однако для улучшения читаемости кода лучше использовать вариант `sym = 'K'`.

| dec | hex  | val   | dec | hex  | val | dec | hex  | val | dec | hex  | val |
|-----|------|-------|-----|------|-----|-----|------|-----|-----|------|-----|
| 000 | 0x00 | (nul) | 032 | 0x20 | □   | 064 | 0x40 | @   | 096 | 0x60 | ‘   |
| 001 | 0x01 | (soh) | 033 | 0x21 | !   | 065 | 0x41 | A   | 097 | 0x61 | a   |
| 002 | 0x02 | (stx) | 034 | 0x22 | ”   | 066 | 0x42 | B   | 098 | 0x62 | b   |
| 003 | 0x03 | (etx) | 035 | 0x23 | #   | 067 | 0x43 | C   | 099 | 0x63 | c   |
| 004 | 0x04 | (eot) | 036 | 0x24 | \$  | 068 | 0x44 | D   | 100 | 0x64 | d   |
| 005 | 0x05 | (enq) | 037 | 0x25 | %   | 069 | 0x45 | E   | 101 | 0x65 | e   |
| 006 | 0x06 | (ack) | 038 | 0x26 | &   | 070 | 0x46 | F   | 102 | 0x66 | f   |
| 007 | 0x07 | (bel) | 039 | 0x27 | '   | 071 | 0x47 | G   | 103 | 0x67 | g   |
| 008 | 0x08 | (bs)  | 040 | 0x28 | (   | 072 | 0x48 | H   | 104 | 0x68 | h   |
| 009 | 0x09 | (tab) | 041 | 0x29 | )   | 073 | 0x49 | I   | 105 | 0x69 | i   |
| 010 | 0x0A | (lf)  | 042 | 0x2A | *   | 074 | 0x4A | J   | 106 | 0x6A | j   |
| 011 | 0x0B | (vt)  | 043 | 0x2B | +   | 075 | 0x4B | K   | 107 | 0x6B | k   |
| 012 | 0x0C | (np)  | 044 | 0x2C | ,   | 076 | 0x4C | L   | 108 | 0x6C | l   |
| 013 | 0x0D | (cr)  | 045 | 0x2D | -   | 077 | 0x4D | M   | 109 | 0x6D | m   |
| 014 | 0x0E | (so)  | 046 | 0x2E | .   | 078 | 0x4E | N   | 110 | 0x6E | n   |
| 015 | 0x0F | (si)  | 047 | 0x2F | /   | 079 | 0x4F | O   | 111 | 0x6F | o   |
| 016 | 0x10 | (dle) | 048 | 0x30 | 0   | 080 | 0x50 | P   | 112 | 0x70 | p   |
| 017 | 0x11 | (dc1) | 049 | 0x31 | 1   | 081 | 0x51 | Q   | 113 | 0x71 | q   |
| 018 | 0x12 | (dc2) | 050 | 0x32 | 2   | 082 | 0x52 | R   | 114 | 0x72 | r   |
| 019 | 0x13 | (dc3) | 051 | 0x33 | 3   | 083 | 0x53 | S   | 115 | 0x73 | s   |
| 020 | 0x14 | (dc4) | 052 | 0x34 | 4   | 084 | 0x54 | T   | 116 | 0x74 | t   |
| 021 | 0x15 | (nak) | 053 | 0x35 | 5   | 085 | 0x55 | U   | 117 | 0x75 | u   |
| 022 | 0x16 | (syn) | 054 | 0x36 | 6   | 086 | 0x56 | V   | 118 | 0x76 | v   |
| 023 | 0x17 | (etb) | 055 | 0x37 | 7   | 087 | 0x57 | W   | 119 | 0x77 | w   |
| 024 | 0x18 | (can) | 056 | 0x38 | 8   | 088 | 0x58 | X   | 120 | 0x78 | x   |
| 025 | 0x19 | (em)  | 057 | 0x39 | 9   | 089 | 0x59 | Y   | 121 | 0x79 | y   |
| 026 | 0x1A | (eof) | 058 | 0x3A | :   | 090 | 0x5A | Z   | 122 | 0x7A | z   |
| 027 | 0x1B | (esc) | 059 | 0x3B | ;   | 091 | 0x5B | [   | 123 | 0x7B | {   |
| 028 | 0x1C | (fs)  | 060 | 0x3C | <   | 092 | 0x5C | \   | 124 | 0x7C |     |
| 029 | 0x1D | (gs)  | 061 | 0x3D | =   | 093 | 0x5D | ]   | 125 | 0x7D | }   |
| 030 | 0x1E | (rs)  | 062 | 0x3E | >   | 094 | 0x5E | ^   | 126 | 0x7E | ~   |
| 031 | 0x1F | (us)  | 063 | 0x3F | ?   | 095 | 0x5F | _   | 127 | 0x7F | ␣   |

5: Фрагмент таблицы ASCII

Таких таблиц кодировок несколько, а может, даже и несколько десятков. Разные операционные системы и разные приложения используют разные кодировки, например, в русскоязычной версии ОС Windows по-умолчанию используется cp1251, в то время как в командной строке этой же ОС используется cp866. Файлы можно сохранить в Unicode или ANSI. UNIX-подобные ОС, такие как Linux и Mac OS X обычно используют UTF-8

или UTF-16, а более ранние операционные системы и интернет пространства в русскоязычном сегменте использовали KOI8-R.

## 10.2 Особенности

Немного вспомнив, что такое символы, переходим к строкам. Строками мы пользуемся с самых первых страниц этого документа: написав в двойных кавычках «Привет, Мир», мы использовали строку, а если точнее, строковый литерал. Строки иногда называют типом данных, но в языке C строка это указатель на последовательно записанный набор символов, поэтому работать с ним можно, как с массивами. Строки в языке C можно описать двумя способами: как указатель и как массив из переменных типа `char`.

Объявление строки как указателя на символы в языке C++ полностью заменили на указатель на константный набор символов, чтобы подчеркнуть неизменяемость литерала. То есть, если в языке C считается нормальной запись `char* s = "Hello";` то в C++ это можно записать **только** как `const char* s = "Hello";` при этом в обоих языках поведение такого указателя будет одинаковым.

Давайте создадим строку в виде массива из `char` назовем ее `string1` и запишем внутри литерал `This is a string!` - это строка. Также создадим указатель, назовем его `string2` и запишем в него литерал `This is also a string!` - это тоже строка. Выведем наши строки в консоль и убедимся, что автор не ошибся и их действительно можно так объявлять.

```
1 char string1[256] = "This is a string!";
2 char* string2 = "This is also a string!";
3 printf("%s \n", string1);
4 printf("%s \n", string2);
5
```

У каждого из способов есть свои особенности. Так, например в *массиве* из переменных типа `char` мы можем изменять символы. Всё работает. Попробовав изменить какой-нибудь символ в `string1`, например, пятый, на символ X можно будет убедиться, что объявленная таким образом строка изменяемая, в отличие от строкового литерала, попытка изменить который приведёт к ошибке во время исполнения программы. Указатель на `char` нам не даёт возможности частично менять содержимое внутри строки, и, получается, представляет собой **immutable string** – неизменяемую строку.

Обратите внимание, что компилятор не считает такую запись неверной и ошибка

```
1 string1[5] = 'X';
2 printf("%s\n", string1);
3
4 string2[5] = 'X';
5 printf("%s\n", string2);
6
```

проявляет себя только во время исполнения программы. Таким образом, становится очевидно, что программисту недостаточно просто убедиться в том, что его код компилируется, но необходимо и проверить его работу во время исполнения. Часто в этом помогают Unit-тесты. Среди нерадивых программистов бытует мнение, что вообще все тесты должны писать специалисты в тестировании, но на примере выше, когда код компилируется, а программа всё равно не работает должным образом, мы видим, что некоторая более тщательная проверка своей работы должна быть выполнена именно программистом, а тестирование - это лишь инструмент, помогающий автоматизировать такие проверки.

```
$ ./program
Hello, world!
Hello, world!
HelloX world!
zsh: bus error ./program
$
```

### 10.3 Строки и функции

Но довольно об ограничениях. Указатели на строки не такие уж бесполезные, мы можем, например, возвращать их из функций. То есть, мы можем объявить тип возвращаемого из функции значения как указатель `char*`, вернуть из нее строку и, например, вывести в консоль. Это открывает перед нами широчайшие возможности по работе с текстами.

```
1 char* helloFunction() {
2     return "Hello!";
3 }
4
5 int main() {
6     printf("%s \n", * helloFunction ());
7 }
8
```

Этот код выведет в консоль ожидаемое приветствие. Параллельно с написанием функции, приветствующей мир, предлагаю изучить некоторые стандартные возможности языка C для работы со строками. Например, специальную функцию, которая призвана выводить строки в консоль: `puts()`; работает она очень похожим на `printf()`; образом, но может выводить только строки, без каких-то других параметров, и всегда добавляет символ конца строки. Также изучим специальную функцию `gets()`; которая призвана считывать строки из консоли и записывать их в переменные.

Функция `gets()`; некоторыми компиляторами признана небезопасной, её использование рекомендуется заменить на `gets_s()`; В C11 и позднее небезопасная функция была вовсе удалена из стандартной библиотеки языка и перестала поддерживаться всеми производителями компиляторов.

Создадим изменяемую строку типа `char[]`, назовём её `name`, передадим эту строку в функцию `gets()`; и выведем на экран результат, полученный из консоли. Это будет очень полезная заготовка для дальнейшего общения с пользователем.

```
1 char name[255];
2 gets(name);
3 puts(name);
4
```

Теперь, мы можем поприветствовать пользователя нашей программы как следует, по имени. В нашей существовавшей функции приветствия внесём небольшие изменения. Создадим строку, в которой будем хранить приветственное слово, и в которую будет дописываться имя пользователя. Применим функцию склеивания строк. Поскольку склеивание - ненаучный термин, будем использовать слово **конкатенация**. И именно это слово подсказывает нам название функции, которую мы будем использовать: `strcat()`;

Для использования функции `strcat()`; необходимо подключить в программу заголовочный файл, содержащий функции, работающие со строками `#include <string.h>`

Функция принимает на вход два параметра - строку, к которой нужно что-то прибавить, и строку, которую нужно прибавить. Логично предположить, что первая строка должна быть изменяемой (то есть являться массивом символов, а не литералом). Функция прибавит все символы второй строки в первую (если в массиве хватит места) и вернёт указатель на изменённую строку. Очень удобно. Запустим наш проект, введем

имя и убедимся что всё сломалось.

```
1 char* helloFunction(char* name) {
2     char welcome[255] = "Hello, ";
3     return strcat(welcome, name);
4 }
5
6 int main(int argc, const char* argv[]) {
7     char name[256];
8     gets(name);
9
10    puts(helloFunction(name));
11    return 0;
12 }
```

Что же случилось? Мы можем возвращать из функции только фиксированные строки, как в предыдущем примере. То есть, получается, нужно писать кейс, в котором содержатся все возможные имена, и оператором вроде `switch()` перебирать все возможные варианты ввода, и описывать все возможные приветствия пользователей, иначе мы устраиваем утечку памяти, создавая болтающийся в воздухе указатель, который никак не удалим? Нет, нас это, естественно, не устраивает. Что делать? Какой бы мы ни создали указатель в функции - он перестанет существовать, как только мы выйдем из области видимости этой функции.

В некоторых случаях может показаться, что никакой проблемы нет, поскольку написанная таким образом программа благополучно поприветствует пользователя, но такое поведение не гарантируется ни одним компилятором и ни одной операционной системой, поскольку возвращаемый таким образом указатель может быть переписан абсолютно любой следующей инструкцией кода. Такое *исчезающее* значение называется **xvalue**.

Выход очень простой: раз указатель не идёт в `int main (int argc, char *argv[])`, надо чтобы `int main (int argc, char *argv[])` дал нам указатель. Добавим в параметры функции указатель на выходную строку, и напишем что для начала сложить строки и положить в локальный массив `strcat(welcome, name)`. Добавим в основную функцию массив `char result[]`, который будет хранить результат и передадим в функцию `helloFunction` аргументы `name` и `result`. А раз функция больше ничего не возвращает, вполне легально сделать её `void`.

```
1 void helloFunction(char* name, char* out) {
2     char welcome[255] = "Hello, ";
3     strcat(welcome, name);
```

```

4     out = welcome;
5 }
6
7 int main(int argc, const char* argv[]) {
8     char name[256];
9     char result[256];
10    gets(name);
11
12    helloFunction(name, result);
13    puts(result);
14    return 0;
15 }

```

Запускаем, и **снова не работает**, да ещё и как интересно, смотрите! Предупреждение, ладно, понятно, мы о нём говорили, но дальше, когда мы вводим имя на выходе получается какая-то совсем уж непонятная строчка, совсем не похожая на приветствие.

```

$ ./program
warning: this program uses gets(), which is unsafe.
Ivan
??:

$

```

А все дело в том, что строк в языке C за время повествования не появилось, и все манипуляции со строками - это довольно сложные алгоритмы работы с памятью и массивами символов. Поэтому, работая со строками, мы должны использовать библиотечные функции, например, есть функция `strcpy()`, которая не просто перекладывает указатель в определенную переменную, а копирует строку.

```

1 void helloFunction (char* name, char* out) {
2     char welcome[255] = "Hello, ";
3     strcat(welcome, name);
4     strcpy(out, welcome);
5 }
6 int main(int argc, const char* argv[]) {
7     char name[256];
8     char result[256];
9     gets(name);
10    helloFunction(name, result);
11    puts(result);
12    return 0;
13 }

```

Если присмотреться, то можно заметить, что все функции работающие со строка-

ми, именно так и делают - запрашивают источник данных и конечную точку, куда данные нужно положить. А кто мы такие, чтобы спорить со стандартными библиотечными функциями? Обратите внимание на то, что функции `strcat()`; и `strcpy()`; возвращают указатель на получившуюся строку. Мы перестали возвращать указатель на получившуюся строку, поскольку никто не гарантирует, что он просуществует достаточно долго, и тут встаёт вопрос о вызывающем функцию контексте, нужен ли этот указатель вызывающему. В случае необходимости, конечно, его можно вернуть. Работа со строками в C до сих пор является очень и очень актуальной темой на программистских форумах, можете удостовериться в этом самостоятельно.

Раз уж заговорили о стандартной библиотеке, рассмотрим ещё пару-тройку интересных функций. Например, сравнение строк: функция `strcmp()`; допустим, я хочу, чтобы именно меня программа приветствовала как-то иначе. Функция возвращает отрицательные значения, если первая строка меньше второй, положительные, если первая больше второй, и ноль, если строки равны. Это функция, которую удобно применять в условиях. Если строки будут действительно равны, мы скопируем в строку с именем слово `Creator`.

```
1 void helloFunction (char* name, char* out) {
2     char welcome[255] = "Hello, ";
3     if (strcmp("Ivan", name) == 0)
4         strcpy(name, "Creator");
5     strcat(welcome, name);
6     strcpy(out, welcome);
7 }
```

Из всех функций для работы со строками чрезвычайно часто используются `atoi()`; и `atof()`; переводящие написанные в строке цифры в численные переменные внутри программы. `atoi()`; переводит в `int`, а `atof()`; во `float`, соответственно.

Существует несколько десятков функций, разнообразно преобразующих одни типы данных в другие и обратно, например, функции `atoi()`; и `atof()`; имеют ответные `itoa()`; и `ftoa()`; . Такие функции пишутся и для высокоуровневых библиотек, например, преобразование строковой записи в понятный компьютеру для соединения IP-адрес.

Для примера объявим переменную `num`, предложим пользователю ввести цифру, естественно в виде строки. Будем принимать ее при помощи небезопасной функции `gets()`; , хотя как мы помним, могли бы и `scanf()`; ; который сразу бы преобразовал строку согласно использованного заполнителя. Заведем переменную `int number` для хранения результата работы функции преобразования. Затем, давайте умножим результат сам на

себя, чтобы убедиться, что это и правда число, причём именно то, которое мы ввели, и выведем окончательное число в консоль.

```
1 char num[64];
2 puts("Enter a number: ");
3 gets(num);
4 int number = atoi(num);
5 number *= number;
6 printf("We powered your number to %d", number);
```

Полный список функций для работы со строками можно посмотреть в заголовочном файле `string.h`. Описание и механика их работы легко гуглится, документации по языку очень много.

## 10.4 Работа с символами

В завершение беседы о строках и манипуляциях с ними, скажем ещё пару слов об обработке символов. Функции для работы с символами содержатся в заголовочном файле `stdlib.h`. Естественно, наша программа может получить от пользователя какие-то значения в виде строк. Не всегда же есть возможность использовать `scanf()`; например, считывание из графических полей ввода или потоковый ввод данных из сети даёт нашей программе значения в виде строк. Стандартная библиотека языка C предоставляет нам функции для работы с каждым символом строки, например:

- `isalpha()`; – возвращает истину, если символ в аргументе является символом из алфавита;
- `isdigit()`; – возвращает истину, если символ в аргументе является цифрой;
- `isspace()`; – проверяет, является ли переданный в аргументе символ пробельным;
- `isupper()`; `islower()`; – находится ли переданный в аргументе символ в верхнем или нижнем регистре;
- `toupper()`; `tolower()`; – переводят символ в верхний или нижний регистр, соответственно.

Можем использовать одну из них соответственно нашей задаче, допустим, пользователь может вводить своё имя как с заглавной буквы, так и всеми строчными. Уравняем оба варианта для нашей проверки одной строкой `name[0] = tolower(name[0]);` а после проверки вернём заглавную букву на место `name[0] = toupper(name[0]);`

и удостоверимся что даже если мы напишем своё имя с маленькой буквы - программа напишет его с большой.

```
1 void helloFunction (char* name, char* out) {
2     char welcome[255] = "Hello, ";
3     name[0] = tolower(name[0]);
4     if (strcmp("ivan", name) == 0)
5         strcpy(name, "Creator");
6     name[0] = toupper(name[0]);
7     strcat(welcome, name);
8     strcpy(out, welcome);
9 }
```

# 11 Структуры

## 11.1 Оператор `typedef`

Иногда бывает удобно создавать для работы понятные или сокращённые названия типов данных. Мы уже рассматривали один из способов создания описаний (9.1), но это были описания, работающие с текстом программы, позволяющие заменять целые фрагменты кода короткими понятными сочетаниями. Оператор `typedef` работает не с текстом программы, а с механизмами компилятора, позволяя создавать так называемые псевдонимы (англ. *alias*) для типов данных. Так, например, можно создать простейший псевдоним для булева типа, который фактически является целым числом. Это делается при помощи ключевого слова `typedef`. Его синтаксис прост, пишем `typedef` название старого типа данных название нового типа, т.е. как мы будем называть его в дальнейшем:

```
1 typedef int boolean;
```

Обратите внимание, что в отличие от директивы `#define` это оператор языка C, а не препроцессора, поэтому в конце обязательно ставится точка с запятой. Написав такой псевдоним мы в любом месте программы можем писать `boolean` вместо `int`, что должно повысить понятность кода для людей, которые будут с ним работать.

## 11.2 Структуры данных

Несмотря на то что язык C создавался в незапамятные времена, уже тогда программисты понимали, что примитивных типов данных недостаточно для комфортного программирования. Мир вокруг можно моделировать различными способами. Самым естественным из них является представление о нём, как о наборе объектно важно помнить, что C - это процедурный язык, в нём не существует объектно-ориентированного программирования.

Существует шутка, что нельзя задавать вопрос программисту на C, является ли структура объектом. Думаю, дело в том, что это введёт C-программиста в бесконечное обдумывание такого вопроса.

Тем не менее, у каждого объекта в мире есть свои свойства. Например, для человека это возраст, пол, рост, вес и т.д. Для велосипеда – тип, размер колёс, вес, материал, изготовитель и прочие. Для товара в магазине – артикул, название, группа, вес, цена, скидка и т.д. У объектов одного типа набор этих свойств одинаковый: все, например,

собаки или коты могут быть описаны, с той или иной точностью, одинаковым набором свойств, но значения этих свойств будут разные.

Сразу небольшое отступление, для тех кто изучал высокоуровневые языки, такие как Java или C#, в C отсутствуют классы в том виде в котором вы привыкли их видеть. Для объектно-ориентированного программирования был разработан язык C++, который изначально называли Си-с-классами.

Так, для работы с объектом нам необходима конструкция, которая бы могла агрегировать и инкапсулировать различные типы данных под одним именем – так появились **структуры**. Т.е. структура данных - это такая сущность, которая объединяет в себе несколько примитивов или других структур. Для примера, создадим структуру «простая дробь». В программировании существуют дробные числа и представлены они типами float и double. Но это десятичные дроби. Мы же будем описывать обыкновенную, простую, смешанную дробь (ту, у которой кроме числителя и знаменателя есть ещё целая часть).

Для *описания структуры* используется ключевое слово `struct` и название структуры. Далее в фигурных скобках описываются переменные, входящие в структуру. В нашем примере это будут целая часть, числитель и знаменатель. У этих переменных не гарантируются инициализационные значения, т.е. мы ничего не присваиваем им изначально, *это просто описание*, которое говорит компилятору о том, что когда в коде встретится инициализация нашей структуры, для её хранения понадобится вот столько памяти, которую нужно разметить для хранения вот этих переменных.

```
1 struct fraction {
2     int integer;
3     int divisible;
4     int divisor;
5 };
```

Для сокращения записи опишем новый тип данных, назовём его **дробь**. Далее будут приведены два равнозначных способа сокращения записи структур: первый способ создаёт псевдоним для уже существующей структуры, а второй создаёт псевдоним для структуры прямо в момент её описания, поэтому и саму структуру можно не называть, а обращаться к ней только через псевдоним:

```
1 // alias for existing struct
2 typedef struct fraction Fraction;
3
4 // alias-defined structure
5 typedef struct {
```

```

6     int nat; // natural (integer)
7     int num; // numerator
8     int den; // denominator
9 } Fraction;

```

Обычно доступ к переменным внутри структуры осуществляется привычным для высокоуровневых языков способом - через точку. Есть одно исключение, но об этом чуть позже. Создадим три переменных для хранения двух дробей, с которыми будем совершать операции, и одну для хранения результата. Инициализируем переменные какими-нибудь значениями. Опишем целочисленные значения, опишем делимое для обеих дробей и опишем делитель для обеих дробей. Для простоты будем использовать простые дроби:  $1\frac{1}{5}$  и  $-1\frac{1}{5}$ .

```

1 int main(int argc, const char* argv[]){
2     Fraction f1, f2, result;
3     // f1 = -1 | 1 /5
4     f1.nat = -1;
5     f1.num = 1;
6     f1.den = 5;
7     // f2 = 1 | 1 /5
8     f2.nat = 1;
9     f2.num = 1;
10    f2.den = 5;
11    // result = 0
12    result.nat = 0;
13    result.num = 0;
14    result.den = 0;
15 }

```

### 11.3 Работа со структурами

Внутри функций структуры данных можно передавать как по значению, так и по ссылке. Опишем функцию, которая будет выводить нашу дробь на экран. В эту функцию мы можем передать нашу структуру по значению. Т.е. внутри каждого вызова функции мы будем создавать копию структуры типа дробь, и заполнять её теми значениями, которые передадим в аргументе. Вывод каждой дроби на экран будет зависеть от ряда условий. Именно эти условия мы и опишем внутри функции вывода на экран. Если делимое равно нулю, то у дроби надо вывести только целую часть, если же делимое не равно нулю и целая часть равна нулю – выводим только дробную часть.

Пишем: если делимое не равно 0 то вступает в силу следующее условие – если целая часть равна нулю, то печатаем дробь следующим образом: число, значок дроби, число, где числа это делимое и делитель, соответственно.

```

1 void frPrint(Fraction f) {
2     if (f.num != 0)
3         if (f.nat == 0)
4             printf("%d / %d", f.num, f.den);
5         else
6             printf("%d | %d / %d", f.nat, f.num, f.den);
7     else
8         printf("%d", f.nat);
9 }

```

Проверим, насколько хорошо мы написали нашу функцию, для этого вызовем ее и передадим туда значения наших дробей. Обратите внимание, что в строках оператора `printf()`; в конце нет ни пробелов ни переходов на новую строку, это сделано на случай, если у нас будет необходимость вставить информацию о дроби в какой-то текст или отобразить в составе уравнения, например.

```

$ ./program
'-1 | 1 / 5'
'1 | 1 / 5'
0

```

Выглядит неплохо, для полноты картины не хватает только научиться выполнять с этими дробями какие-нибудь действия. Для примера возьмём что-то простое, вроде умножения. Передадим во вновь созданную функцию значения наших двух дробей и указатель на структуру, в которую будем складывать результат вычислений. Назовем нашу функцию `frMultiply()`; передадим туда необходимые аргументы и немного вспомним математику. Для того чтобы перемножить две дроби нам надо привести их к простому виду, т.е. лишить целой части, а затем перемножить числители и знаменатели. Для перевода в простой вид опишем функцию `frDesinteger()`; в которую будем передавать адреса первой и второй дроби, чтобы изменять не копии, а сами созданные в `int main (int argc, char *argv[])` дроби. То есть параметрами этой функции должны стать указатели на структуры.

Аналогичным образом, кстати, можно написать функцию инициализации дроби значениями, чтобы не было необходимости для каждой дроби кроме строки объявления писать ещё три строки с инициализацией.

Чтобы не перепутать локальные структуры в функции и указатели на внешние структуры, доступ к полям внутри указателей на структуры получают не при помощи точки, а при помощи специального оператора доступа к членам указателя, который выглядит как стрелка (`->`).

```

1 void frDesinteger(Fraction *f) {
2     if (f->nat == 0) return;
3     int sign = (f->nat < 0) ? -1 : 1;
4     if (f->nat < 0)
5         f->num = -f->num;
6     f->num = f->num + (f->nat * f->den);
7     f->nat = 0;
8 }

```

Аналогично и `result` для функции `frMultiply()`; является указателем, а значит мы будем записывать результат не в локальную структуру, а непосредственно в ту структуру, которую мы объявили в `int main (int argc, char *argv[])` и передали её адрес нашей функции.

```

1 void frMultiply(Fraction f1, Fraction f2, Fraction *result) {
2     frDesinteger(&f1);
3     frDesinteger(&f2);
4     result->num = f1.num * f2.num;
5     result->den = f1.den * f2.den;
6 }

```

Теперь можем выводить результат умножения на экран. Для этого вызовем нашу функцию `frMultiply()`; , в которую передадим две начальные дроби и адрес результирующей дроби. Затем вызовем функцию печати `frPrint()`; и передадим туда в качестве аргумента нашу результирующую дробь.

```

./program
Before:
    f1: -1 1/5
    f2: 1 1/5
result: 0
After:
result: -36 / 25

```

Запустив программу, убедимся что всё работает корректно. Полученных знаний нам хватит практически для любых операций со структурами. Полный листинг программы умножения дробей в приложении С

## 12 Файлы

В предыдущих разделах мы познакомились почти со всеми существующими в языке C типами данных, как примитивными, так и ссылочными. Довольно подробно рассмотрели работу почти всех операторов языка. Пришло время выйти за пределы программы (хотя бы в части хранения данных) и поговорить о взаимодействии программы с операционной системой, а именно - о чтении и записи в файловую систему компьютера. Файловая система *любого* компьютера - это **структура**. Для языка C файл - это тоже структура. Структура, хранящая данные о положении курсора в файле, его название, буферы, флажки и прочие свойства. Файлы делятся на два основных типа - текстовые и бинарные. Мы рассмотрим работу с текстовыми, поскольку работа с бинарными практически ничем не отличается, а текст всё-таки проще воспринимается.

С места в карьер, опишем переменную, хранящую указатель на структуру «файл». Вся основная работа будет проходить через неё. Для того, чтобы присвоить этой переменной указатель на какой-то реальный файл будем пользоваться функцией `fopen()`; , которая возвращает указатель на адрес в памяти. Функция принимает в качестве аргументов имя файла в двойных кавычках и режим его открытия.

```
1 FILE *f;
```

Основных используемых режимов шесть:

- чтение,
- запись,
- добавление,
- двоичное чтение,
- двоичная запись,
- двоичное добавление.

### 12.1 Запись

Функции **записи и добавления** *создают* файл в случае его отсутствия. А функция **записи** *стирает* файл, если он существует и не пустой. Итак создадим текстовый файл с каким-то неожиданным названием, вроде `filename.txt`, и скажем нашей программе, что нужно будет его создать, если его не существует, перезаписать, если существует, а дальше мы будем в него записывать данные, то есть режим открытия будет "w". Имя

файла в аргументе может быть как полным (абсолютным), вроде `C` : `FILE.TXT` тогда файл будет создан в корне диска `C`, или `/home/user/file.txt`. Также имя файла может быть и относительным, каким мы его указали сейчас, это значит, что файл будет создан в той папке, в которой запускается наша программа.

```
1 f = fopen("filename.txt", "w");
```

В случае, если файл не найден или по какой-то причине не создан, в переменную `f` запишется нулевой указатель, поэтому перед тем, как начать работу с файлом, нужно проверить, смогла-ли программа его открыть, для этого запишем условие, что если в наш указатель записался нулевой указатель, то дальнейшее выполнение функции `int main (int argc, char *argv[])` не имеет смысла.

```
1 if (file == NULL) return 1;
```

Если всё хорошо, можем записывать в файл данные. Для записи в файл есть несколько функций, мы воспользуемся самой простой и очевидной: `fprintf()`; в неё в качестве первого аргумента обязательно нужно передать указатель на файл, в который мы собираемся писать, а дальше можно использовать как знакомый нам `printf()`; со всеми его удобствами, заполнителями, экранированными последовательностями и дополнительными аргументами. После того как мы закончили запись в файл его необходимо закрыть, вызвав функцию `fclose()`;

```
1 fprintf(f, "Hello, files! %s", "we did it!\n");
2 fclose(f);
```

Запустив проект, посмотрим что у нас получилось. В терминале не будет никакого вывода, но если перейти в проводник, то можно увидеть, что в папке проекта появился файл `filename.txt`, в котором написано наше содержимое. Открывается такой текстовый файл любым обычным блокнотом

## 12.2 Чтение

Отлично, программа теперь умеет сохранять результаты своей работы в файлы, то есть у нас появилось ещё одно средство вывода информации, помимо терминала. Теперь давайте рассмотрим не менее важную тему, а именно - чтение из файла. Ведь тогда мы сможем не просто иначе выводить информацию, но и сохранять её на жёстком диске от запуска к запуску программы, а значит хранить какие-то настройки или промежуточные значения. Для этого нам нужно совершить несколько несложных действий с применением неожиданно знакомой функции `fscanf()`; чтобы прочитать форматированные значения из файла:

- создать массив `char []`, для примера, назовем его `word`;
- нужный файл открыть в режиме чтения;
- при помощи функции `fscanf ()`; считать из файла некоторую строку, которую положим в этот массив.

Далее выведем в консоль строку которую прочитали, для этого воспользуемся привычной нам функцией `printf ()`; а затем выведем пустую строку. Запустим нашу программу и увидим, что в консоль вывелось слово `Hello`, функция `fscanf ()`; отлично отработала, прочитав все символы из файла до пробела.

```
1 char word[256];
2 f = fopen("filename.txt", "r");
3 fscanf(f, "%s", &word);
4 printf("%s\n", word);
```

Но сколько данных читать? Как узнать, что достигнут конец файла? Для этого придумали функцию `feof ()`; (англ. file: end of file) возвращающую ноль, если конец файла не достигнут, и единицу если достигнут. Опишем цикл, который выведет в консоль все полученные `fscanf ()`; строки из нашего файла. То есть, мы циклически пройдемся по всему файлу, пока не будет достигнут его конец и будем выводить считанные строки в консоль:

```
1 char word[256];
2 f = fopen("filename.txt", "r");
3 while (!feof(file)) {
4     fscanf(f, "%s", &word);
5     printf("%s ", word);
6 }
7 printf("\n");
8 fclose(f);
```

## 13 Распределение памяти

Этот раздел находится в конце книги, но не по важности. Сильная сторона языка C (и, как следствие, C++) не только в возможности работать с указателями, но и в возможности самостоятельно управлять выделяемой памятью внутри программы. В языках высокого уровня данная возможность зачастую скрыта от программиста, чтобы по случайности программа не привела к зависанию среды виртуализации, по неосторожности не попыталась воспользоваться всей возможной оперативной памятью или не сломала операционную систему. Итак, как мы уже знаем, все переменные всех типов как-то хранятся в памяти, и до этого момента нас устраивало, как операционная система нам эту память выделяет. Но, пришло время взять бразды правления в свои руки. Процесс **выделения памяти** для программы называется **memory allocation**, отсюда и название функции, которая выделяет память и пишет в предложенный идентификатор указатель на начало этой области.

### 13.1 `void* malloc(size);`

Функция `void* malloc(size);` принимает в качестве аргумента размер выделяемой памяти. Как видим, функция возвращает довольно необычный на первый взгляд тип: `void*`, то есть область памяти будет зафиксирована, но не размечена. То есть это будет просто некоторая пустая область из `size` байт.

Чтобы иметь возможность в этой области хранить значения нам нужно её подготовить для хранения этих значений – разметить. Например, мы уверены, что будем складывать в нашу область памяти какие-то целые числа типа `int`. Для этого при вызове функции нам надо использовать синтаксис приведения типа полученной области. Мы знаем, что каждая переменная типа `int` хранится в памяти в четырёх байтах. Например мы создаем указатель на некоторую область состоящую из 123 байт – таким образом будет выделена память для 123 байт, но она никак не будет размечена. А при помощи оператора приведения типа мы скажем компилятору, что нам необходимо не только выделить некоторую область памяти, но и поделить её на ячейки размера `int`, и каждой такой ячейке дать свой адрес.

```
1 void* area = malloc(123);
2 int *array = (int*) malloc(123);
```

Здесь важно отметить, что именно такое выделение памяти, как на второй строке, не имеет особого смысла, поскольку 123 не делится на четыре нацело, поэтому у такой области памяти будет не совсем хорошо определённый хвост (последние три байта).

Современные компиляторы действуют немного умнее, чем пишет программист, поэтому, скорее всего, будет выделено чуть больше памяти (обычно, это объём в байтах, равный степеням двойки), то есть для нашего случая, 128. Но рассчитывать на это и строить на этом логику программы не стоит, во избежание досадных ошибок. Итак, как узнать, сколько нужно выделить памяти и при этом не запутаться. Для этого не обязательно знать размеры всех типов данных языка, для этого придумали оператор `sizeof()`.

Интересно, что `sizeof()` - это именно оператор, хоть и выглядит как функция.

Он возвращает размер переменной (типа данных) в байтах. Мы напишем `sizeof(int)` и умножим его на десять, таким образом мы выделим память в размере 40 байт, или под 10 целочисленных переменных. Приведением типа мы разметим выделенную область под хранение переменных типа `int`. Фактически, мы сделали то же самое что и описание массива типа `int` при помощи записи объявления массива с использованием квадратных скобок.

```
1 int *area = (int*) malloc(sizeof(int) * 10);
2 int array[10];
```

Помните, мы говорили про арифметику указателей? Вот это - то самое место, где она нам поможет понять, что вообще происходит. Давайте реализуем два массива: привычным нам способом и при помощи динамического выделения памяти. Для реализации массива нам понадобится его размер, определим его как константу `SIZE`. Заменяем в объявленном массиве `10` на `SIZE` и заполним этот массив какими-нибудь значениями. И напишем второй цикл для вывода этого массива в консоль.

```
1 const int SIZE = 10;
2 int *area = (int*) malloc(sizeof(int) * SIZE);
3 int array [SIZE];
4 int i;
5 for(i = 0; i < SIZE; i++)
6     array[i] = i * 10;
7 for(i = 0; i < SIZE; i++)
8     printf("%d ", array[i]);
```

Добавим пустую строку и сделаем тоже самое со вторым массивом, который мы инициализировали как область памяти. В этом виде данный код явно демонстрирует что мы можем одинаково работать и с массивами, объявленными привычными способами и с динамически выделенной областью памяти. Для более наглядной разницы, при заполнении и выводе в консоль второго массива, воспользуемся арифметикой указателей.

```

1 puts("");
2 for(i = 0; i < SIZE; i++) area[i] = i*10;
3 for(i = 0; i < SIZE; i++) printf("%d ", area[i]);
4
5 for(i = 0; i < SIZE; i++) *(area + i) = i * 10;
6 for(i = 0; i < SIZE; i++) printf("%d ", *(area + i));

```

Напомню, мы реализовали массив `area` вручную. то есть выполняем почти те же операции, которые выполняет компилятор при реализации массива `array`. Практически, разложили синтаксис на базовые операции. Очевидно, что это знание открывает нам возможность распределять память в нашем приложении для любых типов данных, в том числе и сложных, таких как структуры.

Если применить оператор `sizeof()` к указателю на локальный массив (объявленный через квадратные скобки), вернётся размер массива, а если применить этот оператор к динамически выделенному массиву (объявленному через оператор `malloc()`) вернётся размер указателя (8 байт для 64-хразрядных операционных систем).

Для того, чтобы каждый раз не пересчитывать размеры переменных вручную, особенно это актуально для строк и структур, используют оператор `sizeof()`, который возвращает целочисленное значение в байтах, которое займёт в памяти та или иная переменная.

### 13.2 `void* calloc(n, size);`

Функция `malloc()`; резервирует память для нашей программы, но делает это весьма просто, вместе со всеми теми случайными переменными, которые могут там храниться. Если в коде из предыдущего подраздела убрать заполнение массива `area`, с большой долей вероятности, увидим в консоли непонятные ненулевые значения.

Важно, что выделенная память «не гарантирует нулевых значений» в ячейках, то есть значения вполне могут оказаться нулевыми, но это не гарантируется.

Для того чтобы гарантированно очистить вновь выделенную область памяти используют функцию `calloc()`; - clear allocate, которая не только выделит нам память, но и очистит содержимое. Поскольку функция не только выделяет память но и очищает её, считается, что она работает медленнее, чем `malloc()`. Синтаксис её весьма похож, только размеры необходимой области памяти передаются двумя аргументами - первый - сколько элементов, второй - какого размера будут элементы.

```
1 int *area = (int*) calloc(SIZE, sizeof (int));
```

В остальном же выделенная область памяти ничем не будет отличаться от выделенной с помощью `malloc()`;

### 13.3 `void free(ptr); void* realloc(ptr, size);`

По окончании работы с областью памяти надо её освободить, чтобы операционная система могла её использовать по своему усмотрению. Делается это при помощи функции `free()`. Если не освобождать память после использования - велика вероятность того, что мы, например, в каком-то цикле будем выделять себе место под какую-то структуру, и рано или поздно съедем всю память. Неприятно может получиться. Такие ситуации называются «утечками памяти» и могут возникать по огромному количеству причин, особенно во встраиваемых и многопоточных системах.

```
1 free(area);
```

Одной из основных задач программиста является недопущение таких утечек.

И напоследок: довольно часто возникают ситуации, когда нам нужно придать нашей программе какой-то динамики, в этом случае мы можем изменить размеры уже выделенного блока памяти. Например, расширить наш массив, добавив туда пару элементов. Это делается при помощи функции `realloc()`; в которую мы должны передать указатель на область памяти, которую хотим изменить, и размеры новой области памяти в байтах. При помощи этой функции области памяти можно как увеличивать, так и уменьшать, но в этом процессе есть довольно много особенностей, которые выходят далеко за пределы основ языка.

```
1 int newsize = SIZE + 10;
2 area = realloc(area, newsize * sizeof(int));
3 for(i = 0; i < newsize; i++) *(area + i) = i * 10;
4 for(i = 0; i < newsize; i++) printf("%d ", *(area + i));
```

Большинство компиляторов выделяют память блоками, размеры которых обычно равны какой-то из степеней двойки, поэтому при объявлении или изменении области памяти в 20 байт, скорее всего (но это не гарантировано, поскольку не регламентируется стандартом) будет выделена область в 32 байта, или если мы объявим 70 байт, то скорее всего будет выделено 128. То есть при работе с областями памяти не стоит ожидать, что они будут даваться нашей программе подряд. Это приводит нас к беседе о фрагментации памяти и других интересных явлениях, также не являющихся основами языка.

## 14 Итоги

Спасибо, за внимание и интерес, проявленный к этой книге. В тринадцати коротких главах мы узнали как устроена практически любая программа изнутри, научились работать с памятью и указателями, узнали основные принципы и механизмы работы программ на уровне операционной системы. Заглянули внутрь привычных синтаксических конструкций, немного заглянули, что делают и что скрывают от программистов среды виртуализации и фреймворки. Продолжайте изучать технологии и не теряйте веру в себя, когда что-то не получается с первого раза.

# Приложения

## А Полный листинг программы программы терминального калькулятора

```
1 #include <stdio.h>
2
3 int main (int argc, char *argv[]) {
4     float first;
5     float second;
6     float result;
7     int operator;
8
9     printf("Enter first operand: ");
10    scanf("%f", &first);
11    printf("/nEnter 1 for (+), 2 for (-), 3 for (*), 4 for (/): ");
12    scanf("%d", &operator);
13    if (operator == 4) {
14        do {
15            printf("/nEnter second operand: ");
16            scanf("%f", &second);
17        } while (second == 0);
18    } else {
19        printf("/nEnter second operand: ");
20        scanf("%f", &second);
21    }
22    switch (operator) {
23    case 1:
24        result = first + second;
25        break;
26    case 2:
27        result = first - second;
28        break;
29    case 3:
30        result = first * second;
31        break;
32    case 4:
33        result = first / second;
34        break;
35    default:
36        printf("Unknown operator\n");
37        return 1;
38    }
39 }
```

```

40     printf("Result is: %f \n", result);
41     return 0;
42 }

```

## В Полный листинг программы вычисления среднего арифметического

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void printArray(int* array, int length) {
5      int i;
6      for (i = 0; i < length; i++)
7          printf("%d ", array[i]);
8      printf("\n");
9  }
10
11 float average(int* array, int length) {
12     float result = 0;
13     int i;
14     for (i = 0; i < length; i++)
15         result += *(array + i);
16     return result / length;
17 }
18
19 int main(int argc, const char** argv) {
20     #define ARRAY_LENGTH 10
21     int arr[ARRAY_LENGTH];
22     int i = 0;
23     float result = 0;
24     while (i < ARRAY_LENGTH) {
25         printf("Enter value %d:", i);
26         scanf("%d", &arr[i]);
27         i++;
28     }
29     printf("Our array is: ");
30     printArray(arr, ARRAY_LENGTH);
31     printf("And the average is: %f \n",
32         average(arr, ARRAY_LENGTH));
33     return 0;
34 }

```

## С Полный листинг программы умножения дробей

```

1  #include <stdio.h>

```

```

2 #include <stdlib.h>
3
4 typedef struct {
5     int nat; // natural (integer)
6     int num; // numerator
7     int den; // denominator
8 } Fraction;
9
10 void frPrint(Fraction f) {
11     if (f.num != 0)
12         if (f.nat == 0)
13             printf("%d / %d", f.num, f.den);
14         else
15             printf("%d %d/%d", f.nat, f.num, f.den);
16     else
17         printf("%d", f.nat);
18 }
19
20 void frDesinteger(Fraction *f) {
21     if (f->nat == 0) return;
22     int sign = (f->nat < 0) ? -1 : 1;
23     if (f->nat < 0)
24         f->num = -f->num;
25     f->num = f->num + (f->nat * f->den);
26     f->nat = 0;
27 }
28
29 void frMultiply(Fraction f1, Fraction f2, Fraction *result) {
30     frDesinteger(&f1);
31     frDesinteger(&f2);
32     result->num = f1.num * f2.num;
33     result->den = f1.den * f2.den;
34 }
35
36 int main(int argc, const char* argv[]){
37     Fraction f1, f2, result;
38     // f1 = -1 | 1 /5
39     f1.nat = -1;
40     f1.num = 1;
41     f1.den = 5;
42     // f2 = 1 | 1 /5
43     f2.nat = 1;
44     f2.num = 1;
45     f2.den = 5;
46     // result = 0

```

```
47     result.nat = 0;
48     result.num = 0;
49     result.den = 0;
50
51     printf("Before:\n");
52     printf("    f1: "); frPrint(f1); printf("\n");
53     printf("    f2: "); frPrint(f2); printf("\n");
54     printf("result: "); frPrint(result); printf("\n");
55     puts("After:");
56     frMultiply(f1, f2, &result);
57     printf("result: "); frPrint(result); printf("\n");
58 }
```