Информатика. Семинары 1-й курс.

Бабичев С. Л.

7 сентября 2020 г.

Содержание

Здесь размещены материалы, используемые для занятий по курсу информатики 1 года обучения МФТИ. Бóльшая часть материала рассчитана на обычные группы, часть материала предназначена для «продвинутых» групп. Так как лекционный материал на разных факультетах отличается, в материалах будут даны краткие сведения и из лекционного курса, необходимые для изложения материала. Часть материала перекрёстно используется автором на курсе «Алгоритмы и структуры данных» ТехноСферы Mail.ru/МГУ.

Введение

Если вы попали в эту аудиторию, значит вы успешно сдали вступительные экзамены и хорошо разбираетесь (на школьном уровне, конечно) в физике и математике. У меня есть надежда, что вы все окажетесь в состоянии изучить с моей точки зрения более простой предмет — информатику. Я в курсе, как информатику преподают в школе. Школьная информатика скорее похожа на уроки труда в моё время, то есть вам дают в руки какие-то инструменты и учат делать что-то типа табуреток — с одной стороны, достаточно практично, с другой — безумно скучно. На информатике в школе вы учитесь, в основном, работе с одним из инструментов — компьютером. Думаю, что не ошибусь, если скажу, что больше половины времени вы готовили на компьютере разнообразные документы — презентации, ролики, рисовали картинки. К сожалению, при всём своём уважении к школьному образованию не могу сказать, что вы занимались именно информатикой. Умение набирать красивый текст я никак к информатике отнести не могу, это просто полезный житейский навык, который вам пригодится в будущем.

А ведь информатика — наука об информации — является частью математики, прикладной математики. Настоящая информатика (за рубежом она называется *Computer Science*) изучает информацию и методы её обработки. Краеугольным понятием информатики является алгоритм, то есть строго определённый порядок действий по обработке информации. Для записи алгоритмов используют формализованные языки — языки программирования. Мы с вами будем изучать один из наиболее простых языков, на котором написаны триллионы строк кода — язык Си. На этом языке мы и будем реализовывать необходимые вам в будущем алгоритмы.

1 Алгоритмы

1.1 Что такое алгоритмы

Если говорить кратко и неформально, то *алгоритм* — это последовательность команд для некоего *исполнителя*, которая обладает рядом свойств:

- **полезность**, то есть умение решать поставленную задачу;
- **детерминированность**, то есть каждый шаг алгоритма должен быть строго определён во всех возможных ситуациях.
- **конечность**, то есть способность алгоритма завершиться для любого множества входных данных
- **массовость**, то есть применимость алгоритма к разнообразным входным данным.

Любой алгоритм заключается в обработке *входных данных* с целью получения *выходных данных*. В процессе обработки алгоритмы могут использовать *промежуточные данные* и часто бывает удобным, чтобы эти данные были каким-либо образом *упорядочены*, образуя *структуры данных*. Вам (это относится к ФУПМ) предстоит достаточно сложный курс алгоритмов и методов вычислений, на котором вы более формально сможете изучить свойства алгоритмов и доказать их корректность. Наша же с вами цель — и более простая и более сложная одновременно. Мы должны научиться реализовывать эти алгоритмы на языке программирования `Си`, применяя при этом подходящие структуры данных.

Понятия *алгоритм* и *структуры данных* тесно связаны. Перефразируя классика, можно сказать: мы говорим алгоритмы — подразумеваем структуры данных, мы говорим структуры данных — подразумеваем алгоритмы. Невозможно создать хороший алгоритм, опираясь на неподходящие структуры данных.

Каждый алгоритм для своего исполнения (ещё говорят *вычисления*) требует от исполнителя некоторых *ресурсов*. *Программа* есть запись алгоритма на формальном языке.

Одну и ту же задачу зачастую можно решить несколькими способами, несколькими алгоритмами, которые могут отличаться использованием ресурсов, таких, как *элементарные действия* и *элементарные объекты*. Например, исполнитель алгоритма *компьютер* использует устройство *центральный процессор* для исполнения таких элементарных действий, как сложение, умножение, сравнение, переход и других, и устройство *память* как хранителя элементарных объектов целых и вещественных чисел. Способность алгоритма использовать ограниченное количество ресурсов называется *эффективностью*.

А пока давайте введём несколько важных терминов

1.2 Исполнители алгоритмов

Алгоритмы применяются, конечно, не только в информатике. Но именно в информатике важно, чтобы действия алгоритмов были строго регламентированы. Поэтому для *исполнителя* должны быть точно определены все возможные операции, которые он может производить и все возможные данные, которые могут участвовать в этих операциях. Исполнителем достаточно низкого уровня является центральный процессор компьютера. Операции, которые он может производить, достаточно мелкие по нашим меркам, а данные, с которыми он работает достаточно своеобразны и их разнообразие невелико. Более подробно с этим исполнителем вы познакомитесь во втором семестре, а пока в качестве исполнителя будем использовать языки программирования.

1.3 Языки программирования

Я почти уверен, что все из вас знают язык `Pascal` или хотя бы слышали о нём. Можно ли считать этот язык способным описывать алгоритмы? Конечно. Вместо того, чтобы оперировать элементарными операциями такими, как «сравнить две ячейки памяти; если первая больше второй, то выбрать следующую команду для исполнения в ячейке 135; присвоить третьей ячейки значение, извлечённое из второй ячейки; перейти к ячейке 128;» можно написать

```
if a > b then c := b;
```

Другой популярный язык — `Python`, о котором вы тоже, скорее всего, что-нибудь слышали. Считается, что ему учиться легче, чем многим другим языкам. Не уверен, что если под знанием языка понимать написание сложных программ, то это утверждение верно, но для совсем небольших — строк на 100-200 — он вполне может подойти. Те же самые действия, которые мы проводили только что, иллюстрируя `Pascal`, на `Python` смотрятся очень похоже:

```
if a > b:
   c = b;
```

Кстати, и на `Си` мы видим нечто подобное:

```
if (a > b) c = b;
```

Мы видим, что для формулировки своих мыслей (если так, то делаем вот так) все приведённые выше примеры очень похожи и отличаются мелочами — здесь нужно слово `then`, здесь — знак двоеточия `:`, здесь — нужно окружить выражение скобками, но суть у всех этих способов выражения своей мысли одна и та же. Выскажу крамольную мысль: все современные языки достаточно мощны для того, чтобы решать достаточно большое количество задач и выбор

того или иного определяется либо привязанностью конкретного человека или конкретной группы людей либо наличием в конкретном языке неких средств, наиболее подходящих для решения конкретной задачи. Всё это справедливо, пока мы рассматриваем небольшие задачи, до 10-20 тысяч строк кода. Для более крупных проектов или специализированных проектов приходится выбирать языки, предназначенные для этой цели. Например, даже небольшие задачи, которые требуют большого количества вычислений, писать на `Python` я не стал бы, так как тот же самый алгоритм, реализованный на `Си`, может исполняться в 50 раз быстрее. Одно дело, если программа на `Си` исполняется одну секунду, а программа на `Python` — 50 секунд. Другое дело, если программа на `Си` исполняется час или сутки. В этом случае реализовывать алгоритм на `Python` просто бесполезная трата сил и времени. Какие-то языки идеальны для разработки программ одиночками, но имеют массу недостатков, если попытаться с их помощью реализовать совместный проект, другие языки приходится изучать в коллективе, но это позволяет писать программы в миллионы строк.

В любом случае:

> Язык программирования — инструмент для записи алгоритмов неким формализованным образом с целью их исполнения некоторым исполнителем.

> Нет понятия *лучший язык программирования*. Есть понятие: *этот язык программирования хорошо подходит для решения данной задачи*.

1.4 Элементарные типы данных и операции над ними

Любой язык программирования в конце концов отображает свои типы данных и свои операции на те, что доступны компьютеру, то есть именно компьютер будет исполнять то, что мы запрограммировали. Например, в `Pascal`, тип данных `integer` обычно отображается на 16-битные элементы в компьютере, операция `+` над объектами, имеющими такие типы данных в соответствующий набор машинных команд (почему в набор, а не в одну? Узнаете во втором семестре).

К элементарным типам данных обычно относят 8, 16, 32, 64-битные целые числа (последние могут быть элементарными в языке программирования, но не элементарными на конкретном компьютере), символы, которые представляются своим кодом и вещественные числа. Про вещественные числа мы будем много говорить чуть позже. Соответственно, имеются операции, манипулирующие этими элементарными типами данных (точнее сказать, манипулирующие объектами, имеющими элементарные типы данных, но мы не будем уж настолько формалистами).

Человеку удобно, чтобы операции записывались не в виде чисел, как требуют компьютеры, а в виде строк символов, `+,-,*,/,...`.

1.5 Управляющие структуры

Во всех программах, за исключением тривиальных, при исполнении алгоритма в какие-то моменты времени приходится принимать решения: при этом условии исполнить этот фрагмент кода, а при этом — этот. Другой случай: пока значение этой переменной не станет равным нулю, исполнять заданный фрагмент кода. Мы можем разделить управляющие структуры на несколько типов: выбора, цикла, перехода. Каждая из таких управляющих структур записывается на языке программирования существенно проще, чем на машинном языке. Мы с вами можем рисовать блок-схемы для каждой управляющей структуры, но разные языки предоставляют нам разный набор таких управляющих структур, поэтому рисовать их мы будем, когда перейдём к конкретному языку программирования.

1.6 Сложность алгоритма.

Если мы спросим у специалиста по алгоритмам, какая сложность у предложенного им алгоритма, он задаст встречный вопрос: а какую сложность вы имеете в виду? Если требуется реализовать алгоритм в виде схемы вычислительного устройства, реализующего конкретную функцию, то *комбинационная сложность* определит минимальное число конструктивных элементов для реализации этого алгоритма. *Описательная сложность* есть длина описания алгоритма на некотором формальном языке. Один и тот же алгоритм на различных языках может иметь различную описательную сложность. Нас, как составителей алгоритма, больше всего будет интересовать *вычислительная сложность*, определяющая количество элементарных операций, исполняемых алгоритмом для каких-то входных данных. Для алгоритмов, не содержащих *циклов*, описательная сложность примерно коррелирует с вычислительной. Если алгоритмы содержат циклы, то такой корреляции нет и нас интересует другая корреляция — времени вычисления от входных данных, причём обычно интересна именно *асимптотика* этой зависимости. Часто термин *сложность* заменяют противоположным по смыслу термином *эффективность*. Говорят «программа, эффективная по вычислительной сложности», «программа, эффективная по памяти», то есть программа, обладающая небольшой вычислительной сложностью или требующая минимальное количество памяти.

2 Язык программирования Си

Язык Си появился на несколько лет позже языка Pascal и быстро вытеснил того из класса задач, который называется системным программированием. Почему мы изучаем именно Си? Почему в школе его практически не изучают? Ну, про школу всё понятно, что изучают в пединституте, то в школе и дают. Хороший ли язык Pascal? Этот вопрос относится и к Delphi, это просто другое, коммерческое, название языка Object Pascal. Вполне. Для большого класса задач он подходит отлично. Хороший ли язык C++? Отличный. На нём сейчас пишется более 90% системного программного обеспечения. Могу ещё похвалить языки Java, C#, Swift, Go, Rust. Многие без ума от Python. Но у нас будет именно Си. Пока, во-всяком случае. Почему?

Этому есть много причин.

- Синтаксис этого языка послужил основой для синтаксиса таких языков, как C++,Java, C#. Изучив Си, достаточно легко переключиться на более новые языки.
- Близок к машине, программы написанный на нём исполняются быстро.
- Очень компактен. Для его изучения не требуется много времени.
- Много готового кода уже написана и она имеется только на Си
- После Си любой язык покажется удобнее...

Можно даже сказать следующее: язык Си настолько близок к современным вычислительным машинам, что в подавляющем большинстве случаев для создания *эффективных* алгоритмов программирования на языке ассемблера не потребуется. Я не хочу сказать, что язык ассемблера не нужен — для *полного* контроля над компьютером он необходим, но почти всегда его использования можно избежать или ограничить его применение отдельными критическими фрагментами.

2.1 Типы данных языка

Как любой исполнитель алгоритмов, язык Си имеет дело и с тем, что он способен обработать (типы данных), и с тем, каким образом он это делает (операции). Примитивные объекты языка можно разделить на две группы — имеющие целочисленные значения и имеющие вещественные значения. Стоп, а где же символы и строки? Где логические значения? Они в Си относятся к целочисленным типам.

2.1.1 Целочисленные типы

Все современные компьютеры представляют числа в двоичной системе счисления (троичная система как-то не прижилась, а десятичная система представления чисел применяется практически только на *мэйнфреймах* — специализированных чрезвычайно дорогих компьютерах для совместимости со старыми программами). Во всяком случае, типы данных языка `Си` явно позволяют манипулировать битами в своём представлении.

У всех целочисленных типов имеются два свойства: количество битов в представлении объекта данного типа и наличие или отсутствие знака.

Про количество битов вроде бы всё ясно. Сколько разных значений можно представить 8-ю битами? Это простейшая комбинаторная задача: имеется 8 пронумерованных предметов, каждый из которых может быть или белым или чёрным. Сколько различных комбинаций предметов существует? Так как предметы (биты) независимы друг от друга, количество комбинаций есть произведение чисел возможных комбинаций для каждого предмета, то есть $2^8 = 256$. То есть 8-ми битный тип данных способен закодировать 256 различных состояний.

Что есть знаковый тип? Он должен представлять и положительные и отрицательные значения и нуль. Поэтому эти 256 значений мы разделим пополам — половину отдадим на представление отрицательных значений и половину — на представление неотрицательных. Диапазон возможных значений 8-ми битного знакового типа, таким образом, составляет `-128...127`.

Если тип — беззнаковый, то отрицательных чисел в его представлении нет, и эти 128 кодов можно отдать под кодирование положительных значений. Таким образом 8-ми битный беззнаковый тип имеет диапазон `0..255`

Давайте перечислим доступные нам типы:

- `char` — знаковый тип, 8 битов. Мы уже знаем, что диапазон его значений составляет `-128...127`.

  А почему он называется `char`? Потому, что в момент создания `Си` кодировка символов (`characters`) была, в основном, 8-битная и объекты этого типа применялись (да и сейчас применяются) для хранения отдельных символов, точнее, для хранения кодов этих символов.

- `signed char` — в большинстве случаев синоним `char`. Иногда такое поведение можно изменить. И вообще, прилагательное `signed` подразумевается по умолчанию и для других типов данных.

- `unsigned char` — беззнаковый тип, 8 битов. Так как отрицательных значений здесь нет, общий диапазон становится `0..255`.

- `short` — знаковый тип, значения могут лежать в диапазоне `-32768..32767` ($-2^{15}..2^{15}-1$)

- `unsigned short` — беззнаковый тип, значения могут лежать в диапазоне `0..65535` ($0..2^{16}-1$)
- `int` — тип, представляющий «естественные» для данной вычислительной системы целые числа. На старых компьютерах он занимал `16` бит, на большинстве современных — `32` бита, то есть диапазон $-2^{31}..2^{31-1}$ или `-2 147 483 648..2 147 483 647`
- `unsigned int` — то же самое количество бит, что и `int`, но без знака, это или $0..2^{32}-1$ или `0..4 294 967 295`
- `long long int` — знаковый тип, представленный 64 битами с диапазоном от $-2^{63}..2^{63}-1$
- `unsigned long long int` — знаковый тип, представленный 64 битами с диапазоном от $0..2^{64}-1$

Логического типа данных в `Си` нет, все числа способны играть роль логических значений, при этом правила очень просты: истина есть всё, не равное нулю, ложь — всё равное нулю.

Как мы видим, при записи имени типа имеется ряд ключевых слов, которые можно комбинировать. Ключевое слово `short` перед типом данных, *возможно*, уменьшает его длину, а ключевое слово `long`, *возможно*, длину увеличивает. Их можно применять и отдельно, тогда за ними подразумевается слово `int`.

В современных реализациях длина типа `long` часто равна длине типа `int`, но, тем не менее, они считаются различными и не стоит делать предположения, что они равны всегда.

К целочисленным типам относятся также *перечислимые*, их мы рассмотрим позднее (2.11.4)

2.1.2 Вещественные типы

Давайте зададимся вопросом: какое количество информации нужно, для того, чтобы представить число π? А число $\sqrt{2}$? А число 1? На первые два вопроса ответ, вроде бы, простой — сколько бы мы знаков в любой системе счисления не взяли, число точно представить не сможем. А на третий вопрос не столь очевиден. Вроде бы хватит одного знака, хоть в двоичной, хоть в десятичной системе счисления. Правильным ли ответом будет «один бит»? Нет. Он был бы правильным, если бы нам предстоял выбор между, скажем, числами 0 и 1. А если мы выбираем из множества всех вещественных чисел, то, как ни странно, ответ будет «бесконечное количество информации». Вспомним основную формулу количества информации:

$$I = -\log_2 p,$$

где I — количество информации в битах, p — вероятность события «выбрать правильный ответ». Вероятность угадать число на числовой прямой, содержащей бесконечное количество точек равна нулю, следовательно, количество информации при угадывании этого числа равно бесконечности!

Из положения выходят простым образом. В компьютере невозможно представить все возможные вещественные числа. Поэтому берут некоторое подмножество, которое можно закодировать конечным числом бит. Ряд чисел при этом может быть представлен точно, все остальные — приближённо.

В представлении вещественных чисел (а как именно они представляются мы узнаем во втором семестре) важны два параметра: диапазон представления чисел и количество значащих цифр в числе. Компьютер кодирует и то, и другое в двоичном представлении, а нам интересно представление десятичное, поэтому точно ответить на вопрос, сколько десятичных цифр в представлении числа нам доступно, невозможно.

Фундаментальное различие между целыми и вещественными числами на компьютере следующее:

- Целые числа представляют информацию точно, но *дискретность* представления равна единице, поэтому *абсолютная погрешность* представления произвольного числа (при условии попадания числа в диапазон) постоянна.

- Вещественные числа представляют информацию *приближённо*, диапазон представления больше, чем у целых чисел и *относительная погрешность* постоянна.

Имеется два основных вида вещественных чисел:

- `float` — ещё называемые *одиночной точности*. Диапазон представления от примерно $-10^{38} \dots 10^{38}$, количество значащих цифр в представлении от 6 до 7, зависит от числа.

- `double` — числа *двойной точности*. Диапазон примерно от $-10^{308} \dots 10^{308}$. Количество значащих цифр — $16 - 17$.

В ряде реализаций имеются ещё типы `long double` или `extended`, с ещё более широким диапазоном.

Любители и знатоки языка `Python`! Можете преисполниться гордости за свой любимый язык. В этом языке вы можете не задумываться о том, можно ли представить, например, факториал числа 500 в какой-то переменной. В `Си`, увы, это не так. Если вам действительно нужны большие числа, которые не помещаются в `long long` или `unsigned long long`, вам придётся пользоваться *библиотеками*, написанными самостоятельно или кем-то другим. Впрочем, вещественный тип в `Python` имеет те же ограничения, что и в `Си`.

2.1.3 Составные типы

Многообразие всех возможных типов данных, конечно, не исчерпывается основными типами (ещё их называют *базовыми типами*). Любые типы можно объединять друг с другом, образуя:

- *массивы* — несколько объектов одного типа, располагающихся рядом друг с другом и требующих *индекса* для выбора из того, который элемент мы имеем в виду.
- *структуры* (`struct`) — несколько *полей* (*fields*) произвольных типов, требующих *селектора* для идентификации требуемого. Например, мы можем создать структуру `complex`, в которой будут поля `re` и `im`.
- *объединения* `union` — несколько полей разделяют один *адрес*, то есть хранятся в одном и том же месте.

Каждое поле структуры или элемент массива могут быть, в свою очередь, структурой (объединением) или массивом.

Про составные структуры данных мы будем говорить много, но позднее.

2.1.4 Операция `sizeof`

Для любого элемента данных и любого типа данных имеется операция `sizeof` в двух вариантах. Во-первых, операндом `sizeof` может быть любое выражение, в том числе, конечно, и имя переменной, и имя составного объекта. В этом случае операнд необязательно заключать в круглые скобки. Во-вторых, операндом может быть любое имя типа. В этом случае его в скобки заключить необходимо. И в том, и в другом случае `sizeof` возвращает количество минимально адресуемых единиц памяти, *байтов*, требуемых для размещения в памяти операнда или объектов такого типа. Известно, что `sizeof(char)=1`. Остальное зависит от архитектуры компьютера.

```
int sizeint = sizeof(int); // Скорее всего 4
double d = 1.0;
int sized = sizeof d; // Скорее всего 8
```

2.1.5 Литералы

Литерал — элемент записи программы, которые представляет сам себя. Например, числа подходят под определение литерала. Каждый литерал — объект языка `Си` соответствующего типа. Литералы могут иметь *префиксы* и *суффиксы*.

Давайте начнём записывать фрагменты программ, наконец. Но перед этим требуется ввести понятие *комментария*.

Комментарий — часть текста программы, не влияющая на её исполнение.

Комментарии начинаются с символов // и продолжаются до конца строки. Существуют и другие комментарии, начинающиеся с символов /* и заканчивающиеся символами */. Вот эти комментарии могут располагаться на нескольких строках.

Итак, несколько литералов с суффиксами

```
1         // тип int
1l        // буква l - суффикс типа long.
1L        // буква L, и большая и малая - суффикс типа long
1U         // буквы U - суффикс беззнакового типа. unsigned
123UL     // и long, и unsigned
11111ULL  // unsigned long long
123f      // float
123d      // double
123.0     // double
123E17    //  double, суффикс "экспонента", 123 * 10^17
6.02E23   // double, число Авогадро в СГС
'A'       // char, значение равно коду символа A, то есть 65
```

С префиксами бывают только целочисленные литералы. Вот несколько примеров:

```
  0x123   // 0x - шестнадцатеричное представление числа
  0666    // 0 - восьмеричное представление
  '\0x12' // Константа char с кодом 12 в шестнадцатеричной
 системе
  '\0'    // Константа char с кодом 0
```

Строчный литерал начинается с *двойной кавычки*, ей же заканчивается:

```
"This is a string literal"
```

Это — пример литерала-*массива*.

Не путайте одиночные и двойные кавычки, '0'— число, "0" — массив.

Символ \, называемый *бэкслэшем*, в символьных и строчных литералах играет специальную роль — он изменяет значение следующего за ним символа. Вот некоторые константы типа char, которые не имеют печатного представления:

```
  '\n'    // Переход на новую строку
  '\r'    // Переход на начало строки
```

```
'\t'    // Знак табуляции
'\b'    // Возврат на 1 шаг назад
'\\'    // Сам бэкслэш
"Hello\n"  // Слово Hello и переход на новую строку
```

2.1.6 Переменные

Наличие типов было бы бесполезным, если бы мы не могли объявлять переменные соответствующих типов. Правила описания переменных, *синтаксис* описания, в `Си` достаточно прост: за словом, означающим имя типа, следует список *идентификаторов*, *объявляющих* переменные этого типа.

```
int a,b,c;
char querty;
double d;
```

Это — простейший вариант *объявления* переменных. Применяется также слово *декларация*. При объявлении переменной можно сразу же присвоить ей *начальное значение*, или, как говорят *инициализировать*. Одна из неприятных проблем в любом языке программирования — использование *неинициализированных* переменных.

```
int n = 100, m = 500;
double pi = 3.14159265356793;
char delim = '\n';
unsigned mask = 0xFFFF;
```

Позже мы познакомимся с другими способами объявления переменных.

Идентификаторы в `Си` должны состоять только из букв латинского алфавита в любом регистре, знака подчёркивания `_` и цифр, причём цифра не может быть первым символом идентификатора. Вот корректные идентификаторы: `abracadabra`, `N`, `pn766576`, `_`, `__FILENAME__`. А вот некорректные: `Привет`, `1p`.

> Прописные и строчные буквы в идентификаторы различаются (это не `Pascal`). Имена `n` и `N` различны.

2.1.7 Ключевые слова

Некоторые идентификаторы нельзя использовать в качестве имён переменных (и имён функций). Это — *зарезервированные* идентификаторы или *ключевые слова*. В `Си` их сравнительно немного (34 штуки) и их можно перечислить в алфавитном порядке:

`auto`, `break`, `case`, `char`, `const`, `continue`, `default`, `do`, `double`, `else`, `enum`, `extern`, `float`, `for`, `goto`, `if`, `inline`, `int`, `long`, `register`, `restrict`, `return`, `short`, `signed`, `sizeof`, `static`, `struct`, `switch`, `typedef`, `union`, `unsigned`, `void`, `volatile`, `while`.

2.2 Понятие о компиляции и интерпретации

Перед тем, как писать программы, стоит понять, как происходит процесс разработки программ.

Итак, в качестве итога нашего непосильного труда мы получили текст программы на каком-либо языке. Что происходит после того, как мы написали программу? Можно ли *исполнить* написанный нами код, не преобразуя его в что-либо другое? Очевидно, что нет. Подробно процесс преобразования удобного для человека представления программы в представление, удобное для исполнителя *компьютер* мы рассмотрим немного позже, а пока нам нужно ввести несколько терминов, которые мы будем использовать в дальнейшем.

Определение: *Трансляция* — процесс перевода из одного представления программы в другое. Это может быть перевод с `Pascal` на `Си` (существуют и такие трансляторы). Это может быть перевод с `Си` в машинный код. Транслировать (переводить) можно несколькими способами.

Первый способ — переводить предложение за предложением, как это делают переводчики, например, на встрече представителей разных стран. Переводчик выслушал предложение и тут же перевёл его на другой язык. В компьютере это смотрится так: мы ввели строчку программы — она перевелась на машинный язык и тут же исполнилась, и так строчка за строчкой. Этот способ называется *интерпретация*.

Второй способ — прочитать весь переводимый текст и только после этого попытаться перевести его как единое целое. Так переводят книги. То же самое и с программой — чтобы исполнить алгоритм, описанный программой нужно перевести всю программу в машинный код и только после этого исполнить. Этот способ называется *компиляцией*. Похоже на то, что интерпретация удобнее в использовании для написания мелких программок, а компиляция позволит добиться существенно лучшего качества большой программы.

Языки `Си` и `Pascal` обычно используют способ компиляции. Программа, переводящая текст на языке программирования в машинные коды, называется *компилятором*. Мы постоянно будем упоминать этот термин. `Python`, наоборот, использует способ интерпретации. Впрочем, про `Python` нельзя сказать, что программа выполняется строчка за строчкой, так делал в своё время `BASIC`. Программа на `Python` переводится в некий другой *промежуточный* код, который затем интерпретируется *исполняющей системой*. В том числе поэтому программы на `Python` исполняются существенно медленнее эквивалентных программ на `Си`.

2.3 Основные операции

Операций в `Си` достаточно много, большинство из них вполне привычны и легки для понимания. Их можно разбить на несколько групп. Однако, так как *операндами* могут быть объекты разных типов, необходимо ввести понятие *приведения типов* и *приоритеты типов*.

2.3.1 Операция присваивания

Без этой операции трудно представить себе какой-либо язык программирования (хотя такие существуют, как это не странно). Её можно заметить, увидев знак присваивания `=`. Имеется *левая часть* операции присваивания, в которой могут находиться не все конструкции языка, а только так называемые *l-значения*. В правой части могут быть выражения в более свободной форме. Нам сейчас полезно то, что к *l-значениям* относятся *переменные*. О других вариантах *l-значений* мы узнаем попозже.

2.3.2 Приведение типов

Предположим, у нас есть переменная `i` типа `int` и нужно, чтобы её значение было присвоено переменной `d` типа `double`.

Это можно сделать несколькими способами.

1. `d = i;` Это — *неявное* преобразование типа. Язык `Си` допускает такие преобразования, `Pascal` — не допускает.
2. `d = (double)i;` Это — *явное* преобразование типа. Сама переменная `i` при этом, конечно же, не меняется.

В данном примере оба выражения имеют один и тот же смысл.

Преобразование из более длинного целочисленного типа в более короткий производится отбрасыванием «лишних» битов. `(char)256` равен нулю, а не наибольшему из всех возможных `char`.

Преобразование из беззнакового целочисленного типа в знаковый и обратно для типов одинаковой длины производится простым копированием битов. Компьютеру важно лишь содержимое этих битов, а их смысл важен нам, только мы трактуем их как знаковые или беззнаковые величины.

В следующем примере наборы битов, из которых состоят все переменные, одни и те же: `10100001`.

```
char c = -95;
unsigned char uc = c; // c = 256 - 95 = 161
char d = uc; // d = -95;
```

В дальнейшем у нас будут примеры, которые покажут, когда необходимо явное преобразование типов, а когда достаточно неявного.

2.3.3 Типы значений операций

В большинстве операций имеется два операнда и тип получаемого значения зависит от типов операндов. Если типы операндов совпадают, то вопросов нет, тип получаемого значения совпадает с типом операндов. А что делать, если типы не совпадают? Тогда тип результата определяется *старшим* типом операндов. Правил старшинства немного:

1. Вещественные типы старше целочисленных.
2. Беззнаковые типы старше знаковых.
3. «Длинные» типы старше «коротких».

Единственное исключение из этого правила — операции сравнения (2.3.6). Их операнды могут быть произвольных *скалярных* типов, результат — целое число 1 или 0.

2.3.4 Арифметические операции

```
int ia = 10, ib = 20, ic = 30, id = 40;
id = ia + ib; // id <- 30
ia = id - ic; // ia <- 0
ib = id * ib; // ib <- 600
ia = ib / 100; // ia <- 6;
ib = ia % 3; // ia <- 0;
```

Основных арифметических операций в Си немного и их смысл обычно достаточно понятен. Тип результат арифметических операций определяется типом операндов. Сложность может заключаться в том, что результаты арифметических операций могут отличаться от того, что мы ожидаем.

В первую очередь это относится к результатам, которые «вылезают» за разрядную сетку, то есть которые не могут быть представлены нужным типом данных. Например, что получится в результате сложения двух знаковых целых чисел, каждое из которых равно два миллиарда? Четыре миллиарда? Увы. Вспомним про диапазоны типов. Если операнды — 32-битные знаковые целые числа, то результат окажется отрицательным.

Операции над целочисленными операциями длины `n` битов производятся по модулю 2^n, после чего результат трактуется как беззнаковый, если в операции участвует хотя бы один явно указанный беззнаковый операнд и как число со знаком в противном случае.

Во вторую очередь это касается операций деления (`/`) и нахождения остатка (`%`). Если делимое и делитель — неотрицательны, то всё достаточно традиционно, `7 / 3 = 2`, а `7 % 3 = 1`. Всё становится значительно хуже, когда или делимое, или делитель отрицательны. В традиционной математике, точнее сказать, в её подразделе, который называется *модульной арифметикой*, полагается, что если делитель положителен, то и остаток тоже положителен. Соответственно и частное определяется из тождественного выражения

$$a = (a/b) \cdot b + (a \bmod b)$$

В модульной арифметике `-17 mod 10 = 3`, поэтому `-17 / 10 = -2`. Эти же правила применяются и в языке `Python`. Увы, `Си` использует аппаратные особенности компьютеров, на которых он исполняется, а они могут эти правила не соблюдать. Например, на наших любимых компьютерах, иногда называемых персональными (`Intel` ну как же так...), они не соблюдаются:

```
int d = -17 / 10; // d станет равно -1 на  X86 и X64 компьютерах
int e = -17 % 10; // e станет равно -7
```

На компьютерах другой архитектуры это может быть не так!

Будьте внимательны при использовании отрицательных операндов в операциях `/` и `%`. Результаты могут оказаться различными на различных архитектурах ЭВМ.

Обратите внимание ещё на один факт: в отличие от языков `Pascal` и `Python` в Си имеется ровно одна операция деления и это `/`. Тип результат этой операции (как и всех остальных в `Си`) соответствует типам операндов. Никакой отдельной операции `div`, как в `Pascal` или `//`, как в `Python`, в `Си` нет!

2.3.5 Побитовые операции

Все современные компьютеры используют двоичное представление чисел и этим пользуется язык `Си`, включающий много побитовых операций. К ним относятся операции, манипулирующие представлением операндов в двоичном представлении.

Побитовые `&` (*и*), `|` (*или*), `^` *(исключающее или)* и `~` *(не)* применяются к каждому из битов в операнде. Всё происходит строго по законам алгебры логики. Если один из операндов короче другого, он расширяется по правилам

приведения типов. Заметьте, что отрицательные числа представляются в двоичном *дополнительном* коде, а это значит, что *расширение* числа происходит за счёт *распространения* старшего бита. После приведения операндов к одной длине операция производится над каждым из битов в отдельности. Подробнее об этом вы узнаете во втором семестре.

Операция *побитовое и* имеет следующую таблицу истинности:

```
0 & 0 = 0
0 & 1 = 1
1 & 0 = 1
1 & 1 = 0
```

Операция *побитовое или* имеет следующую таблицу истинности:

```
0 | 0 = 0
0 | 1 = 1
1 | 0 = 1
1 | 1 = 1
```

```
  int ia = 3;       //       00000000 00000000 00000000 00000011
  char cb = 5;      //                                   00000101
  int ic = -3;      //       11111111 11111111 11111111 11111101
  char cd = -5;     //                                   11111011
  int ie = ia & cb; // cb -> 00000000 00000000 00000000 00000101
                    // ie  = 00000000 00000000 00000000 00000001
  int if = ic & cd; // cd -> 11111111 11111111 11111111 11111011
```

> Для побитовых операций старайтесь использовать беззнаковые типы данных. Избегайте отрицательных значений до тех пор, пока вы точно не будете понимать, как они представляются в компьютере.

Операция ^ есть операция побитового *исключающего или*.

```
0 ^ 0 = 0
0 ^ 1 = 1
1 ^ 0 = 1
1 ^ 1 = 0
```

Она обладает потрясающим свойством:

```
int a = 123, b = 555;
int c = a ^ b;
// Теперь c^a даст b, a c^b даст a.
```

Операция *побитовое не* имеет следующую таблицу истинности для единичных битов:

```
~0 = 1
~1 = 0
```

Однако не думайте, что если в программе вы напишете `~0`, то получите единицу! Кстати, а что вы получите? Подумайте.

Другие побитовые операции — сдвига влево `<<`, вправо `>>`. Для беззнаковых чисел сдвиг влево на `n` битов эквивалентен умножению на двойку в степени `n`, сдвиг вправо — эквивалентен делению на двойку в степени `n`. Побитовые операции — необходимая и очень важная часть языка. Мы с ними будем много сталкиваться в дальнейшем.

2.3.6 Операции сравнения

Этих операций шесть: `>`, `<`, `>=`, `<=`, `==`, `!=`

В них участвуют ровно два операнда, которые преобразуются к старшему из типов, а результатом операций являются числа целого типа `1`, если условие истинно или 0, если оно ложно.

> Не путайте операции сравнения (`==`) и присваивания (`=`).

```
3 < 5                  // истина или 1
5 == 4                 // ложь или 0
```

Давайте обратим внимание на то, что сравнение вещественных чисел не вполне соответствует интуитивному представлению. Почему? Потому, что вещественные числа принципиально неточны.

```
double d = 1.0;
int r = (d / 3.0) * 3.0 == d; // r может быть 1 или 0!
float f = 1000000000; // Не верьте, число не будет равно 10^9
float g = f + 1; //  g и f равны друг другу!
```

Бороться с этим достаточно сложно. Существует наука «Вычислительная математика», важная часть которой посвящена проблеме неточного представления вещественных чисел на компьютерах. Вы (ФУПМ) будете проходить её на 3-м курсе.

> Сравнение вещественных чисел на равенство принципиально неверно! Лучше всего считать, что все вещественные числа имеют *относительную погрешность*, зависящую от типа. Значения типа `float` имеют погрешность `FLT_EPSILON`, значения типа `double` — `DBL_EPSILON`. Забегая вперёд скажем, что эти константы располагаются в заголовочном файле `math.h`.

2.3.7 Логические операции

Их немного. Это:

- логическое И (`&&`)
- логическое ИЛИ (`||`)
- логическое НЕ (`!`)

> Не путайте побитовые операции `&`,`|` и логические `&&`,`||`.
> `5 | 3 == 7`, а `5 || 3 == 1`
> `3 & 4 == 0`, а `3 && 4 == 1`

Эти операции — *ленивые*. Вычисляется левый операнд, и если оказывается, что результат не зависит от правого операнда, то правый операнд не вычисляется. Например, если в операции `&&` оказывается, что слева результат равен нулю, то каким бы не был правый операнд, результат не изменится, поэтому правый операнд вычислен не будет. Соответственно, единица в левой части операции `||` запретит вычислять выражение в правой части. Это свойство **очень важно** и оно широко используется при вычислениях. Мы много раз будем использовать свойство *ленивости* логических операций, когда начнём работать с массивами.

2.3.8 Тернарная операция. Приоритеты операций

Это — единственная операция в `Си`, имеющая ровно три операнда. Вот пример её использования, вычисляющий наибольшее из пары чисел:

```
a > b ? a : b;
```

Пока то, что мы написали — просто выражение. Для придания примеру какой-либо полезности, можно результат этой операции чему-нибудь присвоить:

```
x = a > b ? a : b;
```

Неопытному глазу не вполне понятно, в каком порядке что выполняется. Пока нам достаточно знать, что приоритетов в `Си` целых 16 штук (многовато) по сравнению с `Pascal`, где 4 штуки (маловато), что приоритеты арифметических операций примерно совпадают с принятыми математически (операции умножения, деления и остатка приоритетнее операций сложения и вычитания), и что они приоритетнее всех логических и побитовых операций и что наименее приоритетная операция — присваивания `=`. Приоритетов много, даже опытные программисты делают неумышленные ошибки в выражениях, содержащих много различных операций.

Для группировки операций в выражении в нужном порядке используйте круглые скобки. Используйте их также в случаях, когда имеются хоть малейшие сомнения в порядке исполнения операций в выражении.

Зная о том, что операция присваивания `=` исполняется в последнюю очередь, мы можем расшифровать вычисление максимума так: «Икс равен (пауза) а больше (повышаем голос, как бы задавая вопрос) б тогда а иначе б.» Тернарная операция — одна из любимых моих операций, позволяющая писать короткие и эффективные программы. Кстати, про неё не забыли и в языках-наследниках `Си` — `C++`, `Java`, `C#`.

2.3.9 Операция *запятая.*

Да-да, имеется и такая операция. Она состоит из нескольких выражений, разделённых запятой и исполняемых слева направо. Результатом операции является самое правое выражение. Эта операция имеет самый маленький приоритет, даже меньше приоритета операции присваивания.

Искусственный пример: `c = (a = 2, b = 3);` Переменной `c` будет присвоено значение 3.

Чуть позже мы увидим полезность такой операции.

2.4 Простейшая завершённая программа

Чтобы с чего-то начать, давайте посмотрим, как выглядит простейшая программа на языке `Си`, которая хоть что-то делает. Мы иногда будем нумеровать строки для того, чтобы на них было удобнее ссылаться.

```
01  #include <stdio.h>
02
03  int main() {
04      printf("Hi again\n");
05      return 0;
06  }
```

Первая строка — указание компилятору на то, что требуется включить файл из *стандартной библиотеки* языка `Си` под именем `stdio.h`. Забегая вперёд скажем, что в этом файле содержится нечто, которое позволяет нам использовать *библиотечные функции.*

Вторая строчка — пустая, мы просто отделяем один законченный фрагмент программы от другого. Такое разделение важно, если вы собираетесь не только писать одноразовые программы, но и намерены читать их в дальнейшем, или если программа пишется несколькими людьми.

Третья строка — *определение* (*definition*) новой функции, под именем `main`, которая возвращает целое значение (`int`) и которая не имеет аргументов `()`.

Тело функции заключается в фигурные скобки (знали ли вы, что группа операторов в фигурных скобках называется блоком?). Закрывающая фигурная скобка находится на 6-й строке и тело функции содержит строки 4 и 5.

Четвёртая строка — вызов функции под именем `printf`, в данном случае имеющий один *аргумент*, строчный литерал. Перед использованием любого имени в `Си` оно должно быть где-то *описано*. *Описание* данной функции содержится в *заголовочном файле* `stdio.h`. Сама функция `printf` по определённым правилам *форматирует* и выводит некий текст на *стандартный вывод*, в данном случае — экран.

Пятая строка — функция `main` заявляет о завершении своего выполнения, возвращая при этом значение `0` в систему. `0` — признак успешного завершения всей *программы*. `main` — особая функция, и как только она завершится, завершится и программа.

Шестая строка — закрывающая фигурная скобка *блока*, начатого в 3-й строке.

Как видите, даже самая простейшая программа требует нескольких понятий, в число которых входит *включение файлов* и *функции*. Более подробно всё это мы изучим далее. А пока у нас есть некий шаблон, позволяющий нам писать крошечные законченные программы.

2.5 Поток управления

В простейших программах, которыми мы с вами будем заниматься весь первый семестр, имеется ровно один исполнитель, *процессор*, который исполняет некоторые действия по изменению значений переменных, изменению порядка вычисления в зависимости от условий и по взаимодействию с окружающей средой (нами). Поток управления состоит из более мелких единиц, *операторов*.

2.5.1 Оператор декларации

В классическом `Си`, созданном Деннисом Ритчи и Кеном Томпсоном и описанном в классической же книге Кернигана и Ритчи «Язык программирования Си»перед тем, как производить какие-либо действия над данными, требовалось все данные *определить* или *задекларировать*. Как это делается мы уже знаем:

```
int a,b,c;
double d = 10.0, e = 3.1415, f = d * e;
```

В стандарте `Си`, вышедшем в 1999 году, который так и называется `C99`, объявления стали полноценными операторами языка и их можно помещать в любое место программы, где разрешён оператор. Не думайте, что переменные всегда надо описывать в начале функции. Наоборот, это — плохой стиль программирования.

Объявления переменных желательно помещать в непосредственной близости от их использования. Тогда их можно инициализировать необходимыми значениями.

2.5.2 Блок

Блок — группа операторов, заключённая в фигурные скобки. Тот, кто писал на языке `Pascal`, хорошо с блоками знаком, там блок есть группа операторов, заключённая в *операторные скобки* `begin` и `end`. С точки зрения синтаксиса языка блок может располагаться в том месте, где допустим единичный оператор и его чаще всего применяют именно для того, чтобы сгруппировать операторы в единое целое.

В отличие от блока языка `Pascal`, блок в `Си` имеет интересное и крайне полезное свойство: все переменные, описанные внутри блока, существуют только внутри блока. Не думайте, что создание и уничтожение переменных потребует серьёзного расхода времени — это не так. Немного попозже мы рассмотрим модель памяти, применяемую в `Си` и вы поймёте, что всё будет происходить чрезвычайно быстро.

2.5.3 Операция присваивания и оператор присваивания

Мы уже использовали присваивание, когда инициализировали переменные. Сейчас время немного более подробно разобраться с многочисленными способами присваивания, существующими в `Си`.

Мы уже рассматривали большое количество различных *операций*, но среди них пока не было *операции присваивания*. Почему мы выделили операцию присваивания в отдельную группу? В `Pascal` ведь нет такого понятия, как *операция присваивания*, но есть понятие *оператор присваивания*. В `Pascal` нет, а в `Си` есть. Язык `Pascal` основан на *операторах*, а `Си` — на *выражениях*.

Предположим, что мы написали

```
a = 5
```

Что это? Пока это — *операция* присваивания. Переменной `a` присваивается значения `5`. Операция присваивания — частный случай *выражения*, а любое выражение в `Си` имеет *тип* и *значение*. В данном случае тип выражения определяется его левой частью, а значение равно 5. Обратите внимание, что точки с запятой после выражения мы пока не поставили! Если мы её поставим, то данное выражение, *операция* присваивания, станет *оператором*. Таким образом в `Си` точка с запятой является *завершителем* оператора, а не *разделителем* между операторами. Кто изучал `Pascal`, тот помнит досаду, когда как только мы ставим ключевое слово `else`, приходится удалять точку с запятой у предыдущего оператора, так как `else` является тоже разделителем истинной и ложной части в операторе `if`.

Хорошо, мы поняли, что произойдёт, если мы *поставим* точку с запятой. А что будет, если мы *не поставим* её? Тогда с выражением можно ещё немного поиграть.

```
b = a = 5
```

Если слева от операции `a = 5` мы поставим переменную `b` и знак равенства, то это будет означать, что мы присвоили переменной `b` значение выражения `a = 5`. Операции присваивания исполняются *справа налево*, поэтому переменная `b` примет значение `5`.

Поиграем ещё.

```
b = (a = 5) + 3
```

Здесь переменной `b` присваивается значение выражения `(a = 5) + 3`, которое, в свою очередь вычисляется как значение выражения `a = 5` плюс значение константы 3, то есть 8. Мы можем играть так и дальше, прекратив игру точкой с запятой в конце выражения, после чего оно превратится в оператор. Такая особенность языка часто приводит к более простым и компактным программам, хотя, если мы заиграемся, читать такое произведение станет трудно.

Но операция присваивания в `Си` имеет много вариантов! Почти все арифметические и побитовые операции имеют вариант с одновременным присваиванием.

Мы можем написать

```
abra_shvabra_cadabra = abra_shvabra_cadabra * 3;
```

и это будет корректно. Но как мы произносим эту строчку, когда читаем программу? Так: «умножим `abra_shvabra_cadabra` на три». Мы опускаем заключение фразы «и присвоим это значение той же самой переменной».

Запись

```
abra_shvabra_cadabra *= 3;
```

более точно отражает алгоритмическую сущность происходящего и, произнося фразу «умножим `abra_shvabra_cadabra` на три» мы читаем именно то, что видим.

```
a += 4;
b = (c *= 2) + 7;
d <<= 1;
e &= 0xFF;
```

Мы видим, что и эти операции присваивания «имеют значение», равное присвоенной величине.

2.5.4 l-значения

В левой части операции присваивания может находиться только то, что способно «принять в себя» какое-то значение. Мы пока знаем только переменные, но слева может находиться и элемент массива и «именующее выражение» (пока мы не знаем указателей и поэтому пропустим уточнение термина). Для того, что может находиться в левой части операций присваивания имеется термин `l-значение` (`l-value`).

> `l-значение` — нечто, существующее в том числе и вне выражения.
> `r-значение` — нечто, существующее только в выражении.

Нельзя написать:

```
3 += a; // Литералы существуют только в выражениях
a + 2 = 4; // Значение a+2  существует только в данном выражении
```

так как в левой части операций присваивания находятся не `l-значения`.

2.5.5 Операции ++ и - -

Мы вынесли эти операции именно сюда, потому, иногда они являются синонимами операций присваивания с присвоением, а иногда имеют отдельный смысл.

Каждая из этих операций имеет два варианта — слева и справа от переменной. Операндом этой операции должно быть `l-value`.

Пре-операции (преинкремента и предекремента) — достаточно просты. Пусть имеется целая переменная `a` и её значение равно 5; `a = a + 1` — операция присваивания, значение которой равно новому значению `a`, то есть 6.

`a += 1` — другая запись той же операции.

`++a` — третья запись той же операции. Результат является `l-value`.

Пост-операции несколько сложнее.

Пусть `a=5`.

Тогда после операции `c = a++;` значение переменной `a` станет 5, а переменной `a` — 6. Переменная `a` увеличивается на 1 в любом случае, но значением любой пост-операции является *старое* значение изменяемой переменной.

> Значение пре-операций есть `l-значение`.
> Значение пост-операций есть `r-значение`.

> Не стоит использовать одну и ту же переменную в выражении несколько раз, если хотя бы одна из операций над переменной есть операция инкремента и декремента.

2.5.6 Оператор `if`

Оператор, без которого в *императивных*[1] языках обойтись невозможно. Вот пример, который вычисляет максимум двух чисел:

```
max = b;
if (a > b)
    max = a;
```

Обратите внимание на отличие от `Pascal`: выражение, которое надо проверить на истинность **обязательно** в `Си` записывается в круглых скобках, что позволяет избавиться от паскалевского слова `then`. При истинности выражения исполняется ровно один оператор, как и в `Pascal`. Если требуется исполнить несколько операторов, их требуется оформить в виде *блока*, то есть заключить в фигурные скобки.

```
min = a;
max = b;
if (a > b) {
   min = b;
   max = a;
}
```

Мы используем *отступы* для того, чтобы показать тому, кто читает программу, что данный блок или единичный оператор исполняется только при соблюдении определённых условий. Сам язык `Си` не заставляет этого делать, в отличие от языка `Python`, но читать и понимать программу становится намного проще, если будет соблюдена *дисциплина программирования*, в данном случае заключающаяся в соблюдении некоторых правил записи программы.

Практика программирования показывает, что всегда стоит использовать форму блока, то есть заключать исполняемый в случае исполнения условия код в фигурные скобки, даже если этот код состоит всего из одного оператора.

```
max = b;
if (a > b) {
   max = a;
}
```

Это тоже дисциплина программирования и следование такому правилу помогает предотвратить много потенциальных ошибок, связанных с тем, что при

[1] Императивные языки программирования основаны на приказах исполнителю выполнить ту или иную команду, возможно, в зависимости от каких-либо условий, то есть описание, *как* решать задачу (`Pascal`, `Fortran`, `C++`, `Java`, `Python`, `Rust`, ...). Декларативные языки программирования описывают `что` надо решать, оставляя на усмотрение компилятора средства решения (`Lisp`, `Haskell`, `erlang`, ...)

добавлении операторов в условную часть можно забыть оформить блок, исказив смысл замысла. Найти такую ошибку в большой программе может оказаться сложным делом. В моей практике мне запомнился случай, когда мой коллега (очень опытный системный программист) искал такую ошибку две недели и нашёл её только с чужой помощью.

Вторая форма оператора `if` содержит *альтернативную ветку*, обозначенную ключевым словом `else`

```
if (a > b) {
   min = b;
   max = a;
} else {
   min = a;
   max = b;
}
```

Отступы у обоих частей хорошо бы делать одинаковыми (если нарисовать блок-схему потока управления альтернативного оператора `if`, то это становится очевидным).

Задача. Даны три числа, $a < b < c$, образующих на числовой прямой 4 интервала. Нужно присвоить переменной r номер интервала (считаем, что интервалы нумеруются с нуля, левый конец отрезка принадлежит интервалу, а правый — не принадлежит, то есть интервал *открыт справа*).

Решение.

```
if (x < a) {
   r = 1;
} else if (x < b) {
   r = 2;
} else if (x < c) {
   r = 3;
} else {
   r = 4;
}
```

Самостоятельно убедитесь, что этот код действительно решает предложенную задачу. Конструкция `else if` обычно применяется для определения попадания некоторого значения в непересекающиеся множества, поэтому отступы всех условий равны. В некоторых языках (`Perl`, `Ruby`, `PL-SQL`) даже имеется отдельное ключевое слово `elsif` для того, чтобы подчеркнуть этот факт.

2.5.7 Оператор `while`

`while` — первый пример того, как можно создавать программы, выполняющие много действий с помощью небольшого количества строк.

Этот цикл напоминает «обыкновенный» оператор `if`, без части `else`. Единственное отличие заключается в том, всё продолжается, пока условие остаётся истинным. Условие истинно — мы заходим внутрь цикла, исполняем все его операторы и, как только мы исполнили последний оператор внутри блока, бежим проверять условие снова. Как только это условие стало ложным, мы в блок с телом не заходим. Вот такой непрекращающийся `if`.

Задача 2.1. Вход алгоритма — целое число `n` до 10^9. Выход — наибольшая число `m` такой, что $3^m \leqslant n$.

Решение. Идея решения: вычислять очередную степень тройки до тех пор, пока она не станет больше `m`, после чего вернуться на единицу назад.

Вторая идея: степень тройки 3^x вычислять можно по индукции, имея вычисленное 3^{x-1}.

```
int pow3 = 1, x = 0, result = 0;
while (pow3 < n) {
    result = x;
    pow3 *= 3;
    x++;
}
```

Ловушка: при умножении на 3 числа больше примерно от 7×10^8 получится отрицательное число. Повлияет ли это на ответ? Подумайте.

При реализации любого цикла убедитесь, что в каждой *итерации* имеется некий прогресс, продвижение переменных к удовлетворению условия. Бесконечный цикл — характерная ошибка, не так редко встречающаяся.

Задача 2.2. Вычислить число e^x с точностью до 6-го знака после запятой для $0 \leqslant x \leqslant 10$ по формуле разложения функции e^x в ряд:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

Решение. Идея решения: будем рекуррентно вычислять очередной член ряда до тех пор, пока он не станет меньше 10^{-7} (это математически нестрого, но достаточно для решения данной задачи).

```
double sum = 1;
int n = 1;
double el = 1;
while ( (el *= x / n) > 1e-7) {
  sum += el;
  n++;
}
```

Мы воспользовались здесь тем, что Си — язык, основанный на выражениях. Значением выражения `el *= x / n` будет `el`, которое в этом же выражении сравнивается с константой 10^{-7}. Этот `el` и есть наш очередной элемент. Этот алгоритм можно сократить ещё больше:

```
double sum = 1;
int n = 1;
double el = 1;
while ( (el *= x / n++) > 1e-7) {
  sum += el;
}
```

Убедитесь, что новый алгоритм эквивалентен старому (не считая завершающего значения `n`).

2.5.8 Оператор `do-while`

Если в операторе `while` условие цикла проверялось *до того*, как войти в блок, то в цикле, начинающемся ключевым словом `do` проверка происходит *после* исполнения всего блока. Задачу по суммированию ряда можно было бы записать и через цикл `do`:

```
double sum = 1;
int n = 1;
double el = 1;
do {
  el *= x / n;
  sum += el;
  n++;
} while (el > 1e-7);
```

Обратите внимание, что ключевое слово `while` всё равно присутствует в операторе. Перенос проверки в конец меняет смысл алгоритма: в цикле `while` мы сначала вычисляли новое значение элемента ряда `el`, и если убеждались, что он больше границы, то прибавляли его к сумме, а в цикле `do-while` мы прибавляли к сумме вычисленное значение `el` и это происходило независимо от того, больше или меньше границы оказывался вычисленный элемент. Уже после того, как мы суммировали его значение с накопителем, мы спохватывались, что элемент вышел за границу и можно завершить подсчёт. Мы прибавили лишний элемент и для получения правильной суммы ряда в данной задаче это не критично. Опыт программирования на Си и подобных языках показывает, что конструкция `do-while` встречается достаточно редко, намного реже «обычного» `while`.

Знающие язык Pascal, обратите внимание на то, что repeat-until, похожий цикл в Pascal повторяется, пока завершающее условие *ложно*, а в Си — пока истинно.

2.5.9 Оператор for

Это — самый мощный оператор цикла. Имеющийся в Pascal цикл for обладает лишь малой частью функционала цикла for языка Си.

Задача 2.3. Найти сумму кубов всех чисел от 1 до заданного n.

Математики, не возмущайтесь, не все программисты знают формулу суммирования кубов натуральных чисел. Дадим им вычислить сумму прямолинейным образом.

```
// Вход алгоритма: n
// Выход алгоритма: sum
int sum = 0, i;
for (i = 0; i < n; i++) {
   sum += i*i*i;
}
```

Обратим внимание на то, что в скобках, идущих после ключевого слова for, имеются *две* точки с запятой. Это — необходимое условие. Эти точки с запятой делят пространство внутри скобок на *три* выражения.

Первое выражение исполняется однократно, как только начинается цикл.

Второе выражение — условие продолжение цикла. Это — логическое выражение, которое проверяется каждый раз, когда *итерация* начинается. В *Си* на этом месте может находиться любое выражение, которое будет считаться истинным, если оно отлично от нуля (вспомним, что в Си нет логического типа данных).

Третье выражение исполняется после того, как будет исполнено *тело* цикла. В данном случае тело — sum += i*i*i; — набор операторов в фигурных скобках, блок. Вместо блока можно разместить ровно один какой-либо оператор, но мы уже знаем, что для того, чтобы избежать плохо отлавливаемых ошибок, лучше оформлять блок, то есть ставить фигурные скобки, даже для единичного оператора.

Применённый нами оператор for мало отличается от того, что мы видели в Pascal. Сила Сишного оператора for в том, что первое, второе и третье выражения независимы друг от друга и мы можем в них производить произвольные действия.

Вспомним задачу по нахождению степени тройки, не превосходящей заданного числа.

```
int pow3, x, result = 0;
```

```
for( pow3 = 1, x = 0; pow3 < n; pow3 *= 3, x++) {
  result = x;
}
```

Используя `for` можно уменьшить вероятность допустить ошибку, поместив изменение переменной, за значением которой мы следим `pow3`, в заголовок цикла. Таким образом мы явно показываем её продвижение к цели (она увеличивается и когда-нибудь достигнет `n`). Программу легче и писать, и читать. Примеров на более тонкое использование всех свойств цикла `for` у нас будет далее в изобилии.

И первое, и второе, и третье выражения можно опустить. Если опустить только первое и третье выражения, то то, что получилось, станет полным эквивалентом `while`. Если опустить второе выражение, то оно будет считаться истинным. `for(;;) {}` — бесконечный цикл. Но как же выйти из бесконечного цикла? Об этом — следующий раздел.

2.5.10 Оператор `break`

Вероятно, это самый простой по форме записи оператор. Он записывается так:

```
break;
```

Если вы смотрели когда-либо бокс, то должны помнить, как рефери постоянно говорит это слово, чтобы разнять боксёров. В `Си` его действие чем-то похоже на действие рефери: прекращается какое-то действие. В `Си` прекращается очередная итерация любого цикла и управление передаётся на первый оператор, следующий за телом цикла. Наличие оператора `break` технически позволяет обойтись только циклом `for(;;)`, тем самым бесконечным циклом, который вызвал у нас небольшое недоумение в предыдущем разделе.

```
for(;;) {
   // ... какой-то код
   if (x > 0) break; // Поняли, что какое-то условие выполнено
   // здесь x <= 0, продолжаем.
}
```

Введение в язык `Си` этого оператора позволило сильно сократить многие алгоритмы. Конечно, можно обойтись и без него, заведя логическую[2] переменную, устанавливая которую и проверяя значение которой можно имитировать выход из цикла. В классическом `Pascal` так и делалось, см. книги Никлауса Вирта по алгоритмам и структурам данных. Никлаус Вирт не любил этот

[2] Я буду и далее говорить о логических переменных, хотя, формально, в `Си` таких переменных нет. Но они есть во всех языках-наследниках `Си` — `C++`, `Java`, `C#`,...

оператор. Всё же заметим, что классический `Pascal` или `Pascal Вирта` был настолько ограничен в своих возможностях, что это был язык не для программистов, а, скорее, для записи алгоритмов. В более современных вариантах языка `Pascal` (про `Delphi`, я думаю, все слышали), `break` стал равноправным оператором.

> Если имеется несколько циклов, вложенных друг в друга, то оператор `break` принудительно завершает только **внутренний** цикл.

`break` применяется также и в операторе `switch`, об этом см. раздел 2.5.12.

2.5.11 Оператор `continue`

Оператор `continue` тоже применяется в циклах, но, в отличие от `break`, он не завершает весь цикл, а завершает текущую итерацию цикла, переходя на новую.

```
for (i = 0; i < 10; i++) {
   // какой-то код
   if (x > 0) continue;
   // Код, который будет пропущен после continue
}
```

Если, положим, при `i` равном 6, сработал `continue`, то управление передастся на третье выражение заголовка `for`, `i++`, `i` станет равным 7, управление передастся на второе выражение и цикл возобновится с новым значением `i=7`.

Опять же, как только мы начнём изучать алгоритмы и их реализацию, мы увидим много примеров использования различных конструкций языка.

2.5.12 Оператор `switch`

Предположим, что нам нужно определить является ли данное число простым числом в диапазоне от 10 до 30 и вернуть нуль в остальных случаях. Незначительность условия позволяет думать, что для реализации данного алгоритма не требуется сложных действий и что явная проверка числа на простоту с помощью каких-либо критериев явно избыточна. С другой стороны, не видно, каким образом создать такое преобразование с использованием простых арифметических действий. А между тем алгоритм прост: если число есть одно из множества `{11, 13, 17, 19, 23, 29}`, то оно — наше. Для подобных случаев имеется оператор `switch`.

```
int ans;
switch (n) {
case 11:
```

```
  case 13:
  case 17:
  case 19:
  case 23:
  case 29:
      ans = 1;
      break;
  default:
      ans = 0;
      break;
  }
}
```

Входом алгоритма является число `n`, выходом — `ans`.

Это — вполне эффективный код, требующий минимальной нагрузки от компилятора и от нас, читающих это.

Оператор `switch` состоит из заголовка (`switch (expr)`), в котором выражение `expr` должно иметь перечислимое значение (то есть иметь целочисленный тип). Случаев `case` может быть любое количество, в них должны фигурировать **константы** и это обязательно (в этом и заключается слабость оператора `switch` языков `Си` и `C++`, в более поздних языках это ограничение убрано). Как только определилось, что значение `expr` совпадает с одной из констант, перечисленных в `метках`, случай, соответствующий данной метке считается активным и начинается исполняться последовательность операторов, следующая за данной меткой. А когда она прекращается? Или она доходит до самого конца оператора `switch`, или она доходит до оператора `break` или исполняется оператор `return`. Необходимость писать `break` после каждого `case` иногда приводит в состояние недоумения: зачем? Ответ прост — когда проектировался язык `Си`, был популярен (гораздо больше, чем сейчас) язык Ассемблера и требовалась простая эффективная конструкция для передачи управления на произвольную точку внутри сложного выражения и этот оператор позволял писать очень эффективный код.

```
double p = 1;
double q = 2.72;
switch (n) {
case 8: p *= q;
case 7: p *= q;
case 6: p *= q;
case 5: p *= q;
case 4: p *= q;
case 3: p *= q;
case 2: p *= q;
```

```
case 1: p *= q;
default:
   break;
}
```

Убедитесь сами, что после исполнения алгоритма для всех `n` от 0 до 8 мы получим $p = q^n$, при этом в машинном коде не будет присутствовать команд, реализующих цикл. Можно сэкономить несколько наносекунд.

Впрочем, в современных языках, даже в тех, на которых язык `Си` сильно повлиял, поведение `case` изменилось и `break` уже ставить не требуется.

Вот ещё один пример:

```
switch (n) {
case 0:
   printf("Ноль");
   break;
 case 1:
   printf("Один");
   break;
 case 2:
   printf("Два");
   break;
 default:
   printf("Не знаю такого числа");
   break;
}
```

Мы редко будем применять этот оператор, однако в системном программировании он весьма популярен. Если в операторе `switch` можно было бы использовать не только константы, он применялся бы гораздо шире, но, как говорится — что есть, то есть.

2.6 Функции

Функция — фундаментальное понятие всех современных языков программирования. В `Си` функция — основной строительный блок, позволяющий упорядочить структуру программы и уменьшить её сложность.

2.6.1 Объявления и определения функций

Перед использованием функции её нужно где-либо *определить*.

Самым простым способом определения функции является помещение её *тела* в *исходном файле* программы где-нибудь *перед* её использованием. Функции расширяют возможности языка, добавлением новых *абстракций*.

Давайте рассмотрим простейший пример. В Си, например, нет операции возведения в квадрат, которая есть, например, в Pascal. Но её очень легко дописать.

```
double sqr(double x) {
    double result = x * x;
    return result;
}
```

Использовать её можно, например, присвоив какой-либо переменной результат её *вызова*:

```
double t2 = sqr(t);
```

Чтобы вызов был успешным, компилятору надо знать как тип возвращаемого значения функции, так и типы всех её аргументов. Если функция *определена* выше *точки вызова*, то у компилятора достаточно информации для того, чтобы проверить правильность её вызова. Если же мы *определяем* функцию где-то ниже точки вызова, то у компилятор может сделать собственные предположения о том, что это за функция и эти предположения могут не совпасть с тем, что он увидит, как только доберётся до её определения. Если невозможно поместить определение функции выше точки её вызова (а это случается не так редко, как может показаться), то мы можем помочь компилятору, поместив *объявление* функции перед точкой её вызова. Для нашей функции sqr объявление будет выглядеть так:

```
double sqr(double t);
```

Объявлений функции в программе может быть любое количество, *определений* — ровно одно.

И определения и объявления начинаются с одного и того же — *заголовка* функции. В определении затем следует *тело* функции, в объявлении — точка с запятой.

В корректной программе для каждой используемой в программе функции заголовки функций должны совпадать и для определений и для объявлений.

Вы заметили, что слова *определение* и *объявление* весьма созвучны? То же самое и в английском языке, *definition* и *declaration*. Чтобы не вносить смуту в неокрепшие умы программистов, был придуман отличающийся от этих слов термин *прототип*.

Объявление функции часто называют *прототипом*.

Этим термином мы и будем пользоваться в дальнейшем.

Если у функции несколько аргументов, то они перечисляются через запятую, каждая со своим типом. Неверно писать:

```
// Неверно!
int max3(int a,b,c);
```

Правильно так:

```
int max3(int a, int b, int c);
```

В этом смысле объявление аргументов функций отличается от объявления переменных внутри функции, где группировка разрешается.

Вот другой пример:

```
double distance(double x1, double y1, double x2, double y2) {
    return sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
}
```

При объявлении функций допустимо не писать имена переменных, перечислив только их типы:

```
double sqr(double);
```

Надо сказать, что имена переменных лучше всё же писать — выбирая осмысленные имена мы можем облегчить понимание программы.

Функцию `distance` можно объявить так:

```
double distance(double x1, double y1, double x2, double y2);
```

А можно и так:

```
double distance(double, double, double, double);
```

Ясно, что первый вариант объявления предпочтителен, так как во втором совершенно неясно, каким образом передаются координаты точек, то ли сначала координаты первой точки по-порядку, затем координаты второй точки, то ли сначала координаты `x` обеих точек, затем координаты `y`.

2.6.2 Передача аргументов в функцию

Как только мы определяем функцию с аргументами, все аргументы считаются переменными, которым присвоены начальные значения. Они приходят из *точки вызова*.

```
int max3(int a, int b, int c) {
  // ...
}


  ...
  int a = 33;
  int b = 77;
  int m = max3(10, a+5, b);  // Здесь - точка вызова.
```

В приведённом примере значением аргумента `a` в функции `max3` будет `10`, значением аргумента `b` будет `38` (сумма значения переменной `a` в точке вызова и `5`), а значением аргумента `c` — `77`.

Функции могут распоряжаться своими аргументами как им заблагорассудится, они могут их изменять, при этом то, что передано в функцию в точке вызова, не изменится.

```
  void foo(int x) {
    x = 10;
  }


  ...
  int a = 33;
  foo(a);
  // Здесь a всё ещё равно 33.
```

Если мы хотим, чтобы функция изменила значение переменной, которую мы в неё передали, нужно воспользоваться *указателями* (2.12.3).

2.6.3 Статические функции

Все функции, которые мы объявляли и определяли до сих пор имели интересное свойство: если, например, в одном файле `f1.c` мы определили функцию `int func(int n) {}`, то ей можно было воспользоваться в другом файле `f2.c`, поместив там прототип этой функции. Однако, если мы не знаем, какие функции имеются в файле `f1.c` и попробовали бы создать функцию `double func();` в файле `f2.c`, то получили бы странное (для нас) сообщение от компилятора при попытке *сборки* программы: `duplicate symbol _func`. В больших проектах полезно разделять функции на те, которыми мы будем пользоваться

только в нашей части проекта, в нашем файле (*внутренние*), и на те, которые мы специально пишем для того, чтобы другие смогли ими воспользоваться (*внешние* или *интерфейсные*). Большое количество внешних функций в программе может сделать невозможным стыковку двух частей большого проекта, созданных в разных файлах разными людьми. К счастью, если перед типом функции в определении или объявлении добавить слово `static`, то этой функцией можно будет воспользоваться только в одном файле, в том, котором она определена.

```
static int func(int n) {
    ...
}
```

Теперь имя `func` можно будет использовать в других файлах проекта.

> Всегда используйте аттрибут `static` для тех функций, которые не планируется делать общими в проекте. Это поможет и вам, компилятор будет способен оптимизировать использование этой функции, зная, что нигде, кроме данного файла она не доступна, и другим, так как вы не *загрязняете пространство глобальных имён.*

2.7 Основы ввода/вывода, функции `printf`, `scanf`

Мы уже знаем, что в `Си` нет встроенного в язык ввода-вывода, какой, например, есть в языке `Pascal`, и нам приходится использовать уже кем-то написанные функции для того, чтобы что-то вывести на экран или что-то ввести с клавиатуры (на большее мы пока и не претендуем). Для этого компилятор предоставляет нам *библиотеку* функций ввода-вывода, которая кем-то уже написана и этот кто-то любезно предоставил нам прототипы всех функций. Для удобства использования эти прототипы (наряду с какими-то структурами данных) он собрал в файл под именем `stdio.h`, который положил в определённое, известное компиляторау место, и который он нам разрешил нам использовать, включив (`#include`) его в нашу программу:

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

Обратите внимания на то, что мы включаем не сам код предложенных нам функций, а только их прототипы, поэтому неточно будет сказать: «подключим библиотеку `stdio.h`», правильнее будет: «подключим интерфейс библиотеки `stdio.h`», так как *тела* всех функций из `stdio.h` уже кем-то откомпилированы и помещены в *библиотеку языка `Си`*, которая автоматически подключается при компиляции (точнее сказать, при *сборке*) любой нашей программы.

2.7.1 Функция printf

Эта функция умеет достаточно многое и достигается это новыми для того времени средствами. Предположим, что вы пишете программу на Pascal и вам нужно вывести 5 переменных, a,b,c,d,e. Это можно сделать так:

```
writeln('a=', a, ' b=', b, ' c=', c, ' d=', d, ' e=', e);
```

Не знаю, как это смотрится для вас, а для меня — не очень хорошо. Трудно сразу заметить, какой вид будет иметь выходная строка. В Си подход совсем другой:

```
printf("a=%d b=%d c=%d d=%d e=%d\n", a, b, c, d, e);
```

По первому аргументу (строке) мы видим общий вид вывода: вместо %d будут подставлены десятичные значения соответствующих переменных.

Первым аргументом в printf идёт *форматная строка* в которой имеется выводимый текст ("a=")и, возможно, несколько *шаблонов* или *спецификаций формата* ("%d"). printf следует по строке слева направо, выводя символ за символом то, что в этой строке находится. Как только он замечает *метасимвол* %, он пока перестаёт выводить текст и начинает собирать *шаблон*. Шаблон заканчивается одной из предопределённых букв, например, буква d означает, что вывод должен производиться в десятичной (decimal) системе счисления. Перед буквой, определяющей формат вывода и тип посланного в printf значения может тоже что-то находиться. Но это давайте посмотрим на примерах, описать **все** возможность printf всё равно не выйдет.

Хотя знак процента является метасимволом, напечатать его тоже можно. Правда, не пройдёт фокус printf("%");, хороший компилятор вас предупредит об ошибке, но вот printf("%%"); уже сделает то, что мы просили — выведет одиночный знак процента.

```
int  i = 123;
char c = 'a';
unsigned u = 256;
unsigned long ul = 4095ul;  // Помните про суффиксы?
long long ll = 65535ll; //
unsigned long long ull = 1024ull;
float f = 123.456;
double d = 12345678.9012345;
// Вывод будем писать в кавычках для того, чтобы видеть и пробелы
printf("i=%d i=%4d i=%04d i=%-4d", i); // "i=123 i= 123 i=0123 i=123 "
printf("c=%c c=%d", c); // "c=a c=66"

printf("u=%u u=%o u=%x u=%X", u, u, u, u);
```

```
// "u=256 u=400 u=ff u=FF"

printf("ul=%ul ull=%ull", ul, ull); // "ul=65535 ull=1024"

printf("f=%f f=%g f=%e", f, f, f);
// "f=123.456 f=1.23456e5 f=123.456"
printf("d=%lf d=%.1lf d=%.7lf d=%10.3lf", d, d, d, d);
//"d=12345678.901235 d=12345678.9 d=12345678.9012345 d=12345678.901"
printf("Result=%.2lf%%", result);
// "Result=10.75%"
```

2.7.2 Функция `scanf`

Для ввода можно использовать функцию `scanf`.

Она создавалась в пару к функции `printf` и весьма на неё похожа.

Первым аргументом у неё выступает форматная строка — то, что функция ожидает на вводе. В ней присутствуют такие же знаки процента, извещающие `scanf`, что ей потребуется ввести число в каком-то формате. Сами форматы в основном совпадают с теми, которые используются в `printf`.

> Функция `scanf` требует, чтобы в неё передавались `l-значения`, то есть адреса, по которым можно присвоить введённое значение.

Несколько примеров:

```
int  i;
char c;
unsigned u;
unsigned long ul;
long long ll;
unsigned long long ull;
float f;
double d;
int code = scanf("%d", &i);
code = scanf("%c %u %lu %llu %f %lf", &c, &u, &lu, &llu, &f, &d);
```

Функция `scanf` возвращает число тех адресов, по которым ей удалось положить значение. А почему она могла не сделать какой-то работы? Например, потому, что мы просили число, а на входе оказалось нечто нечисловое. Может быть, закончился входной файл (в том числе и стандартный ввод).

Пусть на входе имеется строка вида `17/12/2017`. Тогда ввести её можно так:

```
int day, month, year;
code = scanf("%d/%d/%d", &day, &month, &year);
```

Если при этом code окажется равным трём, то всё ввелось успешно.

Функции `printf` и `scanf` настолько сложны и многогранны, что на их полное описание понадобились бы не один десяток страниц. Можно сказать, что они формируют небольшой язык описания форматов. Поэтому мы с ними расстаёмся, но не навсегда. Использовать в дальнейших примерах мы их будем очень часто. Оставайтесь на связи.

2.7.3 Функции `getchar/putchar`

Символы — средство взаимодействия компьютера с пользователями. Именно символы выводятся на экран, именно символы вводятся с клавиатуры. Конечно, символы в компьютере тоже представляются набором битов, *байтами*, и именно эти байты мы и должны сформировать при подготовке вывода информации на экран. Мы уже знаем, что такие функции, как `printf`, позволяют выводить нам произвольные строки символов и они же умеют преобразовывать внутреннее представление чисел в набор выводимых байтов. Каждый выводимый на экран символ имеет свой *код*, например, символ нуля имеет обычно код, равный 48, а символ пробела — код, равный 32. Помнить все коды — неблагодарная задача, компьютер справится с этим куда лучше. Поэтому в `Си` не пишут числовые значения кодов, а заменяют их символьными константами, такими, как `'0'` или `' '` (здесь внутри — знак пробела).

Сама функция `putchar` — одна из простейших функций. Она выводит свой аргумент, заданный кодом символа, в виде символа на экран. В принципе, можно было бы ничего больше не придумывать, а вывод, скажем, чисел писать с помощью функции `putchar`. Но, честно говоря, это было бы довольно скучно. Посмотрим, как она работает.

```
putchar('H');
putchar('e');
putchar('l');
putchar('l');
putchar('o');
putchar('\n');
```

Мы вывели слово `Hello` и перешли на новую строку.

Функция `getchar` делает противоположную работу — она вводит ровно один символ (с клавиатуры). Правда, все современные операционные системы не позволят вам в обычном режиме реагировать на абсолютно все нажимаемые клавиши — многие клавиатуры передают системе несколько символов при нажатии на одну клавишу, а нажатие на другие клавиши, например на `Alt` наша обычная программа вообще не сможет распознать. Как мы потом убедимся, и функция `putchar`, и функция `getchar` при определённых обстоятельствах позволят работать и с файлами, но об этом лучше прочитать там, где пишут про операционные системы. Наше же дело — алгоритмы.

Давайте посмотрим примеры.

```
int c = getchar();
if (c == EOF) {
   printf("End of file reached\n");
} else {
   printf("You just type character '%c' with code %d\n",c,c);
}
```

Не показалось ли вам необычным, что мы завели переменную `c` типа `int`, хотя хотим ввести просто один символ? Да, это действительно необычно, но объяснимо. функция `getchar`, если есть что считывать, возвращает число в диапазоне `0..255`, что позволяет распознать все возможные символы любой из 8-ми битных кодировок, например, `CP1251` или `CP866` (эти кодировки используются для представления кодовых страниц, содержащих русский язык)[3] А как распознать факт, что ввод уже закончился (как говорится, достигнут конец файла стандартного ввода)? Функция `getchar` для этого использует специальное значение, которое не может быть кодом никакого из символов, `EOF`, константу, описанную в файле `stdio.h`. Это и позволяет нашей программе распознать эту ситуацию, не прибегая к другим средствам.

Чтобы показать программе, что конец файла достигнут, при вводе с клавиатуры в операционных системах `Microsoft` требуется нажать `Ctrl/Z` и затем `Enter`, в операционных системах с ядром `UNIX` (Linux, MacOS) — `Ctrl/D` и больше ничего.

2.8 Игра с управляющими структурами

Пусть у нас имеется некая задача, у которой имеются входные данные и которая выдаёт нам какие-то выходные данные. Сколькими способами её можно решить, сохраняя неизменность *спецификации*[4] задачи?

Задача, которую мы будем решать, имеет следующую спецификацию:

Задача 2.4. На вход алгоритма подаётся натуральное число N. На выходе должно быть число M такое, что $2^M \leqslant N < 2^{M-1}$.

Говоря математически, задача заключается в вычислении целочисленного логарифма по основанию 2 от N. Давайте реализуем алгоритм в виде функции. Функцию нужно как-нибудь назвать и имя должно отражать сущность алгоритма. Неплохим именем будет, например, `ilog2`. Первая буква `i` обозначает, что это — целочисленная функция, `int`, для вещественных функций мы

[3] Если вы хотите работать с `UNICODE`, то потребуются особые функции обработки таких строк, которые лежат за пределами нашего рассмотрения.

[4] Спецификация задачи (алгоритма) — совокупность требований, предъявляемых к входным данным и к тому, каким образом выходные данные должны соотноситься с входными.

оставим имя `log2`. *Прототипом* этой функции будет `int ilog2(int n);`

Способ 1. Заведём переменные `down` и `up`, которые будут нам показывать нижнюю и верхнюю границы поиска при соответствующем `m`. Мы будем увеличивать переменную `m` каждый раз на 1, увеличивая вместе с ней переменные `down` и `up` в два раза. Как только окажется, что наше число `n` лежит в границах между `up` и `down`, алгоритм завершится.

Вначале мы должны определить начальные значения переменных. Минимальным значением `m` является 0, поэтому значением переменной `down` должно быть $1 = 2^0$, а переменной `up` — значение $2 = 2^{0+1}$. Алгоритм закончится тогда, когда `n` попадёт в нужные границы между `down` и `up` и выходным результатом станет `m`. Условие продолжения итераций противоположно условию завершения итераций.

```
int ilog2(int n) {
  int m = 0;
  int down = 1;
  int up = 2;
  while (!(down <= n && n < up)) {
    m++;
    down *= 2;
    up *= 2;
   }
   return m;
}
```

Первое, что мы должны сделать, написав реализацию алгоритма — убедиться, что алгоритм корректен, то есть для всех допустимых входных данных он выдаёт правильные выходные. Мы реализовали алгоритм прямолинейным образом, использовав две граничные переменные, которые *синхронно* изменяются вместе с переменной `m` и которые на каждой итерации цикла сохраняют истинность *высказываний*, *предикатов* $down = 2^m$ и $up = 2^{m+1}$. Эти высказывания сохраняют истинность на всё время исполнения алгоритма и являются *инвариантами*. Наличие таких инвариантов и позволяет нам подтвердить корректность алгоритма.

Второе, что требуется сделать, после необходимой части проверки корректности — подсчитать сложность алгоритма. Пока это для нас сложно и непривычно, но давайте попробуем. Немного подробней про нотацию описания сложности алгоритма можно посмотреть в книге, посвящённой алгоритмам. В данном случае *главным параметром* алгоритма является `n`. Нетрудно убедиться, что общее число требуемых итераций совпадает с вычисленным значением `m`, а `m` есть $\lfloor \log_2 n \rfloor$, так что и сложность всего алгоритма будет составлять $O(\log n)$, то есть верхней границей количества операций будет нечто, пропорциональное

$\log n$. Поищем (очень примерно!) коэффициент пропорциональности. Внутри цикла имеется три операции, в заголовке цикла — ещё три (два логических выражения и логическое `ИЛИ`), правда, их вычислительная сложность разная. В первом приближении можно сказать, что коэффициент амортизации (более научное наименование коэффициента пропорциональности в теории сложности алгоритмов) C равен `6` а общая сложность — $(O(6n))$.

Давайте совершенствовать наш алгоритм, попытавшись уменьшить коэффициент амортизации C.

Способ 2. Обратили ли вы внимание на то, что мы производим лишние операции? Например, если `n=10`, то вначале мы вычисляли значение операций сравнения `n >= 1` и `n < 2`, на второй итерации — `n >= 2` и `n < 4`, на третьей — `n >= 4` и `n < 8`, и так далее. Не смущает тот факт, что если оказалось, что истинно высказывание `n < 4`, то уже не окажется истинным высказывание `n >= 4`? Можно ли это использовать для сокращения количества операций в алгоритме? Да, это значит, что переменная `down` нам не нужна совсем и от неё можно избавиться:

```
int ilog2(int n) {
  int m = 0;
  int up = 1;
  while (n >= up) {
    m++;
    up *= 2;
  }
  return m;
}
```

Корректность алгоритма доказать чуть сложнее, но мы уже проделали необходимые рассуждения.

А что произошло со сложностью? С точки зрения предельных функций (ещё раз посмотрите определение O-нотации в разделе **??**, если вы достаточно дотошны) O-сложность не изменилась, как раньше она была $O(\log N)$, так и теперь она осталась $O(\log N)$. Но ведь достаточно очевидно, что алгоритм стал тратить меньше операций на проведение одной и той же работы. Просто количество операций уменьшилось пропорционально для любых значений `N`. Да-да, это тот самый *коэффициент амортизации*. Для алгоритмов, одинаковых по сложности в O-нотации меньшее значение этого коэффициента будет означать более быстрый алгоритм. Но, повторяю, самым главным показателем сложности алгоритма всё же остаётся O-выражение.

Способ 3. Представим, что мы знаем значение функции `ilog2(N)` для какого-то N. Можем ли мы, не производя новых вычислений, быстро определить `ilog2(2*N)`? Да, легко. `ilog2(2*N) = ilog2(N)+1;`. Можно догадаться, что имеет место следующее рекурсивное соотношение:

$$ilog2(N) = \begin{cases} 0, \text{ если } N = 0 \\ 1 + ilog2(N), \text{ если N>0} \end{cases}$$

Любые выражения такого рода легко программируются:

```
int ilog2(int n) {
  if (n <= 1) return 0;
  else return 1 + ilog2(n / 2);
}
```

Не забудем, что операция `n / 2` даёт нам целую часть от деления числа на 2. Деление на 2 на компьютерах с двоичным представлением чисел (поищите другие, и сообщите мне, если найдёте, ладно?) можно произвести существенно более быстрой побитовой операцией сдвига на 1 вправо, правда, только для неотрицательных чисел (а у нас именно такое после проверки).

Для придания изящества окончательному решению добавим любимую мной *тернарную операцию*:

```
int ilog2(int n) {
   return n <= 1 ? 0 : 1 + ilog2(n >> 1);
}
```

Оценим сложность этого алгоритма. По количеству операций ничего нового мы не получили — количество вызовов функции равно возвращаемому значению, то есть $O(\log N)$. А вот по памяти несколько хуже. Пока `ilog2(100)` вызывает `ilog2(50)`, чтобы, получив результат, добавить единицу и вернуть сумму, в момент вызова `ilog2(50)` в памяти находятся *обе* функции. Более того, когда рекурсия не закончится (при условии `n <= 1`), в памяти окажется вся цепочка вызовов, и `ilog2(100)`, и `ilog2(50)`, `ilog2(25)`, `ilog2(12)`, `ilog2(6)`, `ilog2(3)`, `ilog2(1)`. То есть алгоритм стал хуже *по памяти*, её требуется тоже $O(\log N)$, когда до этого требовалось $O(1)$. Уменьшилась и *описательная сложность*, так как программа стала проще. Обратите на это внимание, рекурсивные программы часто имеют меньшую описательную сложность, чем эквивалентные нерекурсивные.

2.8.1 Перевод числа в систему счисления (рекурсивный вариант)

Оперируя числами, мы привыкли к их десятичной записи, однако кто из вас задумывался, что стоит за выводом какого-либо числа на экран?

Задача 2.5. На вход алгоритма подаётся число $n > 0$. Требуется вывести его десятичное представление. Разрешено пользоваться только функцией вывода одного символа `putchar`.

Что значит вывести число? Сказать компилятору, что его нужно вывести? Это в Си невозможно в принципе, в Си нет встроенного ввода/вывода и можно только сделать это с использованием каких-либо функций. Но нам дали немного — нас научили выводить ровно один символ. Справимся ли мы с задачей вывода числа?

Давайте напишем такую функцию. Назовём её `outnum`. Надо придумать её параметры и определить тип возвращаемого значения. С параметрами, как будто, всё ясно — это число целого типа `n`. А что мы ждём в качестве возвращаемого значения? Цель функции состоит только в выводе цифр на экран, никаким переменным присваивать возвращаемое значение не требуется, поэтому типом значения, возвращаемого функцией будет `void` — пусто. Прототип функции, следовательно, будет `void outnum(int n);`

Перед тем, как продолжить написание функции, мы должны вспомнить, что взаимодействие компьютера с нами возможно только с помощью символов. Несмотря на то, что внутри себя компьютер имеет дела с числами, как только касается его взаимодействия с нами, числа исчезают и на поле боя вступают символы. Таким образом нам требуется перевести такое, скажем, число, как 123 в набор из трёх символов — `'1'`, `'2'` и `'3'`. К нашему счастью, все системы кодирования символов числами применяют следующее соглашение — численные значения всех символов от нуля (`'0'`) до девяти (`'9'`) имеют последовательно расположенные коды. Теперь наша задача упрощается, нам для того, чтобы определить последнюю цифру числа, достаточно взять остаток от деления этого числа на 10. И? Что с этим числом делать, ведь, скажем, если мы хотим напечатать десятичное представление числа `123`, нам нужно вывести вначале символ `'1'`, а мы можем найти цифру `'3'`? Есть способ — запоминать все цифры получившегося числа в массиве, который потом можно вывести в обратном порядке. Но можно воспользоваться и рекурсией, задержав вывод последней цифры, пока остальные экземпляры функции не выведут то, что перед ней.

```
void outnum(int n) {
   if (n > 0) {
      outnum(n/10);
      putchar('0' + n%10);
   }
}
```

Может показаться странным, но мы свою задачу выполнили! Как и в любой рекурсивной функции, вызовы функции самой себя не должны продолжаться бесконечно. Здесь мы поставили ограничитель — аргумент должен быть строго больше нуля. Как только аргумент оказывается равным нулю, рекурсия прерывается. Эта функция далека от универсальной, так как она не может напечатать правильный результат, если мы вызовем её как `outnum(0);`. Может быть, заменить `n > 0` на `n >= 0`? Попробуйте, только не обижайтесь на

компьютер, если программа аварийно завершится. Причина в том, что теперь нет способа прекратить рекурсию и программа захватывает всё больше драгоценной памяти из стека.

2.8.2 Быстрое возведение в степень

Здесь — кусочек для математиков.

При вычислении чисел Фибоначчи можно воспользоваться аппаратом линейной алгебры. Введём вектор-столбец $\begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$, состоящий их двух элементов последовательности Фибоначчи и умножим его справа на матрицу $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$:

$$\begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} F_1 \\ F_2 \end{pmatrix}.$$

Для вектора-столбца из элементов F_{n-1} и F_n умножение на ту же матрицу даст:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix} = \begin{pmatrix} F_n \\ F_{n-1} + F_n \end{pmatrix} = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}.$$

Таким образом,

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}$$

Математика закончилась, но подсказала нам, что некоторые алгоритмы можно свести к другим. Окажутся ли они проще или сложнее — мы пока не знаем и определим после разбора нового алгоритма.

Математика сказала нам, что для нахождения $n-$го числа Фибоначчи достаточно возвести матрицу $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ в $n-$ю степень. Давайте решим задачу попроще — возведём какое-либо число в n-ю натуральную степень. Можно ли сделать это за число операций, меньших $n - 1$?

Выясняется, что можно. Намёк: если нам нужно возвести число в степень 128, не требуется 127 умножений, достаточно возвести получающееся число в квадрат семь раз. Из этого можно вывести рекуррентную формулу:

$$x^n = \begin{cases} 1, & \text{если } x = 0 \\ \left(x^{\frac{n}{2}}\right)^2, & \text{если } x \neq 0 \wedge n \bmod 2 = 0 \\ (x^{n-1}) \times x, & \text{если } x \neq 0 \wedge n \bmod 2 \neq 0 \end{cases}$$

Здесь мы можем записать этот алгоритм на языке программирования в виде рекурсивной функции:

```
double mypow(double x, int n) {
  if (n == 0) return 1;
  if (n % 2 != 0) return mypow(x, n-1);
  double y = mypow(x, n/2) * x;
  return y*y;
}
```

Попробуем оценить сложность немного другим способом. Представим степень, в которую мы возводит, в виде двоичного числа, например, степень `25` в виде `11001`. Тогда нечётная степень будет означать, что последний разряд в двоичном представлении степени есть единица и операция `n-1` есть её обнуление. Чётная же степень будет означать, что последний разряд равен нулю и деление такого числа на два есть вычёркивание этого разряда. Каждую из единиц требуется уничтожить, не изменяя количества разрядов и каждый из разрядов требуется уничтожить, не изменяя количества единиц. Сложность обеих операций составляет $O(\log N)$ (докажите), поэтому сложность итогового алгоритма тоже $O(\log N)$.

2.9 Простые массивы

Массив — фундаментальная строительная единица в большинстве языков программирования. Абстрагируясь от деталей реализации, мы можем сказать, что массив — нечто, что позволяет хранить в себе много пронумерованных элементов одного и того же типа, причём время доступа к любому из элементов одинаково. Можно представить массив как ряд стоящих одинаковых внешне шкафов, на которых написаны их номера, а что внутри шкафа — мы можем увидеть, как только подойдём к нему и откроем. Мы недаром выделяем понятие *простой массив* в отдельный раздел, так как в `Си` массивы тесно интегрированы с указателями и более полный разбор всех возможностей и свойств массивов мы отложим до раздела 2.12.3.

Объявить массив можно, например, таким образом:

```
int ar[100];
double z[32];
```

У каждого простого массива есть свойства: имя (`ar` или `z` в нашем примере), тип его элементов (элементы `ar` имеют тип `int`, элементы `z` имеют тип `double`) и количество элементов (100 элементов в массиве `ar`, 32 элемента в массиве `z`).

Обратите внимание на то, что количество элементов должно быть известно в момент компиляции программы, то есть то, что мы записываем в квадратных скобках, как размер массива, должно быть вычислено на этапе компиляции программы, то есть быть *константным выражением*[5].

После создания массива к его элементам можно обращаться `по индексу`.

```
ar[10] = 15;
z[0] = 3.1415926;
```

[5]Вы можете возразить, что в нескольких компиляторах, самым известным из которых является `gcc`, размер простого массива можно задавать и как переменную, но во-первых, это приводит к программам, которые потом не смогут компилироваться, скажем, оптимизирующим компилятором `Intel C`, а во-вторых, поведение программы станет зависеть от размера массива, в частности, массивы, содержащие миллионы элементов таким образом создавать нельзя.

Массив, объявленный `int ar[100]`, содержит ровно 100 элементов, которые нумеруются от 0 до 99. Обращение по индексам вне этих границ приводит к непредсказуемым результатам.

То, что в Си индексы массивов нумеруются с нуля имеет глубокий смысл, который мы поясним в разделе «указатели» 2.12.3.

Кстати, мы сказали, что результаты при обращении к элементам массивов за границей индексов будут непредсказуемы. А как это может проявиться? Последствия могут быть различными. Самым страшным на вид, но, по-сути, самым простым последствием будет то, что ваша программа внезапно прекратит исполнение, выдав какое-либо системное сообщение. Вы такие сообщения часто видите, когда работаете в Windows: «Программа выполнила недопустимую операцию и была закрыта», «Windows ищет причины завершения вашей программы» (в Интернете, ага, *нашей* программы, творческих успехов). Почему самое простое? Да потому, что вы увидели, что ошибка в программе имеется и её можно попытаться обнаружить. Гораздо хуже, когда возникает ситуация, что вы используете индексы за границами массива и при этом «портите» значения переменных, которые находятся рядом с вашим массивом. Ну вот представьте, вы только что присвоили значение `125` переменной `x` и, так как вы ничего другого ей не присваивали, вы убеждены, что оно там и находится. Вдруг в какой-то момент оказывается, что там уже число `-33`. В какой момент значение изменилось, установить очень трудно. А просто вы по ошибке написали `ar[i]=-33;` в тот момент, когда переменная `i` имела значение `100`. Такую ошибку можно искать очень долго, если не воспользоваться специальными *инструментальными средствами* для тестирования программ (например, `valgrind` или `address-sanitizer`). Но это всё достаточно сложно. Давайте пока просто писать такие алгоритмы, которые не могут покинуть внутренности массива.

2.9.1 Операции над массивами

С точки зрения языка Си выражение вида `ar[i]` является `l-value` (см. 2.5.4), а это значит, что оно может использоваться точно в тех же местах программы, где могла бы использоваться обычная переменная — в выражениях, слева в операциях присваивания, как аргументы функций и т. д.

```
int ar[100],i;
...
i = 0;
if (ar[i] < ar[i+1]) {
  int t = ar[i];
  ar[i] = ar[i+1];
  ar[i+1] = ar[i];
```

```
}
```

Что делает данный фрагмент кода? Он обменивает местами элементы `a[i]` и `a[i+1]`, если они расположены в порядке возрастания. Какое условие корректности данного фрагмента? Мы должны быть уверены, что не `i`, не `i+1` не могут находиться вне диапазона `[0..99]`, а это значит, что `i` обязано подчиняться условиям $0 \leqslant i \leqslant 98$.

> В `Си` нет встроенных операций над массивами, как целыми объектами. Для работы с массивами нужно явным образом пройтись по всем его элементам.

По элементам массива проходят обычно с использованием циклов. Цикл `for` предоставляет удобную запись для этого:

```
int i;
for (i = 0; i < 100; i++) {
  ar[i] = 0;
}
```

Обратите внимание на `i < 100;`. Этим мы показываем, что элемент под номером `100` уже лежит за границами массива, то есть в математической нотации, множество индексов есть `[0..100)` или множество закрытое слева и открытое справа.

Предположим, у нас есть два одинаковых по размерам массива из однотипных элементов.

```
int ar1[100], ar2[100];
...
```

Как скопировать один массив в другой? Попробовать `ar2 = ar1;`? Не получится.

А вот так:

```
int i;
for (i = 0; i < 100; i++) {
  ar2[i] = ar1[i];
}
```

уже можно.

2.10 Строки (введение)

Можно со всей откровенностью сказать, что строк как специального типа данных в `Си` нет совсем. То, что имеется в `Си` — зачатки нормальной работы со строками. Строки в `Си` — средство очень низкого уровня. По сути они

представляют собой обыкновенные массивы. Более подробно о строках мы поговорим после изучения указателей. А пока нам требуется понимать и различать строки и одиночные символы. Как мы помним, одиночные символы в Си присутствуют, константы выглядят так: 'A', '9', нечто, что помещается в **одиночные** кавычки и значением этой константы будет ASCII-код символа. Те же самые символы, помещённые в **двойные** кавычки приобретают совершенно другой смысл. У нас появляется **массив** таких символов, причём к концу этого массива невидимо для нас приписывается 8-битный байт с кодом 0.

```
"A" ==  {'A', 0}
"Hello" == {'H', 'e', 'l', 'l', 'o', 0}
```

Раз это массив, то для него определена операция индексации [] и это можно использовать соответствующим образом:

```
putchar("0123456789ABCDEF"[x % 16]); // вывод 16-ричной цифры.
int sz = sizeof("Hello"); // sz = 6
```

2.11 Структуры

Мы уже умеем использовать массивы в нашей деятельности и знаем, что, хотя количество элементов в массиве мы поменять можем, а вот типы элементов — нет. Если нам это действительно нужно, лучше забыть про Си и обратиться к языкам Go или Python. Впрочем, в массивах нам не хватает не только такого достаточно странного функционала. Нам может потребоваться не тотальная смена в непредсказуемое время типов элементов массива, а, скорее, возможность хранить где-то элементы разных типов, собранные в единое целое.

Например, если мы хотим моделировать автомобиль, то описать его одним, пусть даже большим массивом, содержащим элементы одного типа, не удастся.

2.11.1 Что есть структура

> ***Структура*** — тип данных, позволяющий сгруппировать объекты возможно разных типов и работать с ними как с единым целым. Сама структура становится ***объектом***, а её составные части становятся ***подобъектами*** или ***полями***.

Для модели автомобиля структура, например, может состоять из полей num_of_wheels — количестве колёс целого типа, velocity — текущей скорости автомобиля вещественного типа и reg_number типа «массив символов».

Такую структуру можно было бы описать так:

```
struct car_s {
  int num_of_wheels;
  double velocity;
  char reg_number[12];
};
```

У нас появился новый тип данных — `struct car_s`. Теперь мы можем создавать переменные, которые имеют соответствующий тип:

```
struct car_s a,b,c;
```

За ключевым словом `struct` здесь следует идентификатор `car_s`, который называется *тегом* или *ярлыком* структуры, что позволяет создавать неограниченное количество структур различного типа. Впрочем, некоторые считают, что употребление ключевого слова `struct` каждый раз немного громоздко. Возможно, они правы. Во всяком случае, в `C++` ключевое слово `struct` после объявления структуры писать не нужно. В `Си` такое сокращение тоже можно сделать, использовав ключевое слово `typedef`.

```
typedef struct car_s car;
```

После такого объявления у нас появился ещё один тип данных — `car`, который можно использовать везде, где допустимо имя типа.

```
car a, road[1000], *pa = &a;
```

Мы объявили одиночный автомобиль `a`, массив из 1000 автомобилей `road` и указатель на автомобиль `pa`, который инициализировали адресом автомобиля `a`.

2.11.2 Операции над структурами

А что же можно делать со структурами? Во-первых, структура становится полноценным типом данных и над ней можно производить универсальные для всех типов данных операции: копировать, передавать в функции, определять адрес, возвращать значение данного типа из функции. Вполне корректна запись:

```
car some_function() {
  car ret;
  ...
  return ret;
}

...
car t = some_function();
cat copy_t = t;
```

В отличие от языка Pascal структуры являются полноценными объектами языка. В Pascal, например, функция не может возвращать структуру, а в Си, как мы только что убедились, это возможно. Но не увлекайтесь. Возвести в квадрат структуру, даже если она представляет собой комплексное число, не получится. В Python умножение объекта на число даст новый объект — список. В Си такого нет.

А как обращаться к *полям* структуры? Для этого применяется операция обращения к полю структуры, которая выглядит так

```
car ret;
ret.number_of_wheels = 4;
ret.velocity = 10.7;
strcpy(ret.reg_number,"A789AB150");
```

С функцией strcpy, применённой здесь, мы ещё не знакомы, про неё и про подобные функции есть соответствующий раздел 2.14. Пока же мы можем рассматривать поле reg_number просто как массив.

После применения этой операции (она ещё носит названия *квалификация поля*) поле структуры становится неотличимым от обычной переменной, ему можно присваивать соответствующие его типу значения, а если оно имеет элементарный тип, то и операции типа += или ++. А вот с целой структурой такие операции уже не пройдут. Таким образом, нельзя сказать, что, создав структуру, мы создаём тип данных, способный полностью имитировать встроенные. Нельзя написать ret += 10; Это, конечно, ограничение, которое в Си никак не обходится. Если нужно, чтобы новый тип был «как родной» в языке, придётся использовать C++.

2.11.3 Объединения

Сколько памяти требуется для хранения всей структуры? Например, сколько байт будет выделено для объектов следующей структуры:

```
struct some_1 {
   double x;
   int y;
   short z;
   char c;
};
```

Хорошо, а этой:

```
struct some_2 {
  char c;
  double x;
```

```
  short z;
  int y;
};
```

Эти структуры содержат одни и те же элементы, поэтому логично предположить, что и памяти они потребуют одинаковое количество. На самом деле это не так. На большинстве компиляторов (современных!) с установками по умолчанию первая структура займёт 16 байт, а вот вторая — 24 байта. Чудеса? Нисколько. Архитектура современных компьютеров такова, что для быстрейшего доступа к элементам данных требуется *выравнивание* этих элементов на определённые адреса. Например, если поместить объект типа `double` по адресу, не кратному 8, то все операции с ним потребуют больше *процессорных тактов*, то есть программа станет исполняться медленнее — а кому это нужно? Кстати, на некоторых типах процессоров *невыровненные* данные вообще запрещены — хотите пользоваться 32-битными объектами — извольте выровнять их по адресам, кратным четырём. Хотите пользоваться 128-битными объектами — извольте выровнять их по адресам, кратным 16. И так далее. Подробнее об архитектурных особенностях процессоров мы узнаем во втором семестре (ФУПМ).

Таким образом, принимая во внимание выравнивание, мы можем определить *смещение* от начала структуры для каждого из её полей:

```
struct some_1 {
  double x;  // 0
  int y;     // 8
  short z;   // 12
  char c;    // 14
  // дырка 1
};

struct some_2 {
  char c;   // 0
  // Дырка 1-8
  double x; // 8
  short z;  // 16
  // Дырка 18-20
  int y;    // 20
};
```

Иногда нам нужно экономить память, причём очень жёстко. Для этого мы можем использовать другую модификацию структур — *объединение*.

```
union some_3 {
    char c;   // 0
```

```
    double x; // 0
    short z;  // 0
    int y;    // 0
};
```

Все поля в объединении начинаются с одного и того же адреса. Это означает, что если вы положили что-либо в `x` и потом хотите оттуда что-то забрать как `x`, то всё будет в порядке. Вы можете забрать и кусочки представления `x`, например, младшие 8 битов, обратившись к `c`. Что вы там прочитаете — зависит от архитектуры компьютера, но раз вы этого хотите, язык это предоставляет.

```
union some_4 {
  double d;
  unsigned char c[8];
};
```

Таким образом мы можем обращаться к отдельным байтам вещественного представления числа. Другой способ сделать это — использовать указатели (2.12.3).

Ещё один интересный технической приём — использование *безымянных* объединений.

```
struct sample {
   int f1;
   double f2;
   union {
      short f3;
      int   f4;
   };
};

struct sample s;
s.f1 = 1;
s.f3 = 123;
```

В данном примере переменные `f3` и `f4` располагаются на одном и том же месте памяти.

> При операциях с полями `union` будьте внимательны: присвоение одному полю приводит к изменению другого.

2.11.4 Перечисления

Строго говоря, перечисления относятся к целочисленным типам, но их синтаксис ближе к структурам, так что мы их рассмотрим здесь.

```
enum color {
   red, green, blue
};

enum color c1 = red,c2 = green,c3 = blue;
```

Несмотря на атрибутику структур (наличие *тега* `color`), полей у перечислимого нет. То, что описывается внутри, формирует целочисленные поименованные константы. С одной стороны они имеют тип `enum color`, с другой — они могут использоваться в любых операциях на правах целочисленных *констант*. Сами значения констант формируются компилятором автоматически (`red = 0, green = 1, blue = 2`), но их можно назначать и вручную:

```
enum color {
    red = 0, green = 0x100, blue = 0x10000
};
```

Часто бывает намного удобнее использовать для объявления констант именно перечислимые типы, особенно если требуется массовое их объявление.

Если какому-то элементу перечисления (`green = 0x100`) присвоено конкретное значение , то каждому следующему элементу будет присвоено значение на единицу больше, чем предыдущему, но мы в любой момент можем присвоить что-нибудь конкретное (`blue = 0x10000`).

2.12 Память

2.12.1 Автоматическая память

Мы уже наобъявляли большое количество разных переменных и теперь должны знать, что все переменные, объявленные внутри функций (в том числе функции `main()`) называются *автоматическими* и располагаются в *стеке*. Самое полезное свойство такого класса памяти в том, что все переменные, объявленные в *блоке* (группе операторов, заключённой в фигурные скобки `{ }`), автоматически исчезнут, как только блок будет закрыт. Это очень сильно экономит память в реальных программах и позволяет компилятору делать некоторые ловкие трюки. Например, вместо размещения переменной, локальной в блок, в оперативной памяти, стеке, компилятор может выделить ей какой-либо регистр процессора. Если учесть, что скорость работы с регистрами на порядок (да-да, не менее, чем в 10 раз) превосходит скорость работы с оперативной памятью, то очевидно, что мы должны помогать компилятору в этом добром деле.

Объявление в блоке переменной с именем, которое уже существует в *охватывающем* блоке, в Си разрешается и такое объявление *скрывает* внешнее имя. Иногда это полезно, но будьте с этим осторожны.

```
int f(int n) {
   double x = 123;
   if (n > 0) {
      int x = 5; // Здесь мы полностью скрыли double x
      // Это совсем другая переменная со своей памятью
   }
}
```

Автоматические переменные — хорошее дело, но, к сожалению, в любой операционной системе и в любом компиляторе вы столкнётесь с ограничениями, связанными с количеством разрешённой для автоматических переменных памяти. Это связано с тем, что все современные процессоры поддерживают несколько *потоков исполнения*, *нитей*, которые могут исполняться одновременно. Аккуратное использование *многопоточности* может в несколько раз ускорить исполнение алгоритма, но каждый из потоков использует свой стек, который по этим причинам не может быть особенно большим.

Экономно обращайтесь со стеком, не помещайте туда больших массивов. Стек — ценный и невосполнимый ресурс.

2.12.2 Статическая память

Термин *статическая* память намекает нам, что эта память *неподвижна*. Если, например, в функции имеется переменная под именем x, то она может находиться в разных местах оперативной памяти в разные моменты времени. Более того, если функция вызовет саму себя, то переменных x будет столько, сколько раз произошёл *рекурсивный* вызов. В отличие от автоматической памяти, каждой переменной с классом статической памяти будет выделен один единственный адрес на всё исполнение программы (хочется добавить: «до самой смерти»). В статический класс памяти переменную можно отправить двумя способами: сделав её *глобальной*, то есть объявив её вне функций, либо добавив слово static перед её типом.

```
int gv, c[100000]; // Глобальные переменная и массив

int func() {
   int i;
   gv = 0;
```

```
   for (i = 0; i < 100000; i++) {
      gv += c[i];
   }
}

int bar() {
   static int sv; // Статическая переменная
   if (sv == 0) {
      printf("Вызвали функцию bar\n");
      sv = 1;
   }
}

int main() {
   func();
   printf("gv=%d\n", gv);
   int i;
   for (i = 0; i < 100; i++) {
      bar();
   }
}
```

После исполнения этой программы мы увидим

```
gv=0
Вызвали функцию bar
```

> Статические и глобальные переменные не меняют своего адреса во время исполнения программы и инициализируются нулями (если они не инициализированы иначе).

В нашем примере большой массив `c[100000]` инициализирован нулями, поэтому сумма его элементов тоже нулевая. Сообщение о том, что вызвана функция `bar` было выведено на экран только один раз, так как значение переменной `sv` внутри функции `bar` сохраняется между вызовами функции и, однажды став единицей, больше не изменится.

Обратите внимание на то, что переменные `gv` и `c` *видны* во всех местах программы после их объявления, а переменную `sv` видно только внутри функции `bar`. Это означает, что имя `sv` как статической переменной функции `bar` вне функции использовать нельзя, мы *не видим* эту переменную, несмотря на то, что она существует всё время исполнения программы.

Истинно глобальные переменные можно использовать и в других *единицах компиляции*. Для этого достаточно *объявить* их, как *внешние*, добавив ключевое слово `extern`:

```
// Файл main.c
int gv, c[100000];

...
// Файл user.c
extern int gv, c[100000];
```

Если мы не добавим ключевое слово `extern`, то компиляция (точнее, сборка) нашей программы завершится неудачей, так как каждое определение глобальной переменной добавляет имя в *глобальное пространство имён*. В это же глобальное пространство имён добавляются и имена функций (если их не объявить `static`, об этом мы уже знаем из 2.6.3).

> Постарайтесь ограничить применение глобальных переменных. Не загрязняйте пространство имён.

Ещё одна особенность глобальных и статических переменных: их использование сильно усложняет написание параллельных алгоритмов. Впрочем, сейчас вам кажется, что вам это никогда не понадобится. Поживём — увидим.

2.12.3 Указатели.

Наконец-то мы добрались до самого интересного и самого необычного в `Си` — *указателей*. Указатели существовали в нескольких языках до `Си`, но только в `Си` они получили чрезвычайно большую мощность. Если вернуться в 1970-е годы, то тогда практически одновременно появились два небольших языка программирования, претендующих на универсальность, `Pascal` и `Си` (`Pascal` появился на 3 года раньше). Первый `Си` был гораздо несовершеннее нынешнего, но именно на `Си` стали разрабатывать и системные программы, и расчётные программы, и прикладные программы. `Pascal` пережил всплеск, связанный с почившей ныне фирмой `Borland`, но во все времена количество *промышленного* кода, написанного на `Си` во много раз превосходило таковое на любом из диалектов `Pascal`. Убеждён, что если бы при проектировании языка `Си` не изобрели бы указатели с их арифметикой, то `Си` не выжил бы в конкурентной борьбе с другими языками.

Итак, на арену цирка выходят *указатели*.

Определение указателя элементарно:

> **Указатель** — переменная, содержащая в себе адрес какой-то области памяти.

Раз речь пошла об *адресах*, представим унарную операцию `&` — операцию *взятия адреса*. Мы её использовали в функции `scanf` для считывания переменных со стандартного ввода.

```
int v = 5;
int *pv;
pv = &v;
```

Знак звёздочки * перед переменной при её объявлении показывает нам, что переменная будет указателем на тот тип, что находится слева от звёздочки. Например, строчку `int *pv;` следует читать так: объявляем переменную `p`, которая является указателем (`*`) на целое (`int`). Можно написать и `int* pv;`. Рекомендую использовать первую форму записи, прижимая звёздочку к имени переменной. Почему? Вот простой пример:

```
double* a,b,c;
```

На первый взгляд кажется, что мы объявили три указателя (`double *`) на `double`. А ведь на самом деле объявлен один указатель `a` и две простые переменные `b` и `c`. Запись

```
double  *a,b,c;
```

лучше отражает смысл объявлений. Впрочем, новые возможности языка `Си` — объявление переменных в месте их появления, делают такие групповые объявления совершенно ненужными.

Операция взятия адреса `&` может применяться только к тем элементам, которые могут быть *l-значениями*, например, к переменным, элементам массивов и структур. l-значениями не являются литералы, кроме строк и выражения типа `10 * 20`. Впрочем, об этом написано подробнее в 2.5.4.

Итак, указателям можно (и нужно) присваивать какие-то адреса, причём адреса не первые попавшиеся, а адреса кусков в памяти, содержащих значения нужного типа.

```
int v = 5;
double *pv;
pv = &v;  // не стоит так делать.
```

Хороший стиль программирования подразумевает, что любое объявление переменной сто́ит совмещать с инициализацией. Особенно это относится к указателям. Объявление `int *pv;` просто выделяет место в стеке, достаточное для хранения адреса какой-либо целой переменной, но это — просто выделение памяти. В этой памяти может находиться произвольная информация и крайне маловероятно, чтобы эти биты вдруг волшебным образом собрались в нечто, содержащее реальный адрес. Поэтому

При объявлении указателей всегда инициализируйте их либо корректным адресом, либо специальным значением `NULL`, гарантирующим, что по этому адресу ничего нет.

2.12.4 Указатели и функции

Пока не совсем ясно, для чего нам нужны указатели — экая невидаль, какая-то переменная содержит адрес другой переменной — что из того? Хорошо. Давайте напишем функцию `swap`, которая меняет местами значения двух переменных. Она нам пригодится чуть позднее, когда мы займёмся алгоритмами, в частности, при сортировке массивов.

Первый вариант:

```
#include <stdio.h>

void swap(int x, int y) {
   x ^= y;
   y ^= x;
   x ^= y;
   printf("swap: x=%d y=%d\n", x, y);
}

int main() {
   int x = 5, y = 7;
   swap(x, y);
   printf("x=%d y=%d\n", x, y);
}
```

Что выведет эта программа? Увы, не то, что мы ожидаем.

```
swap: x = 7 y = 5
x = 5 y = 7
```

Вспомним про то, что мы говорили об параметров функций: *с параметрами функции можно делать что угодно — это только копии аргументов.* Так что мы, конечно изменили значения переменных `x` и `y` внутри функции, но это были только копии. Распечатывали же мы `различные` переменные, хотя назвали их одинаково: переменная `x` в функции `swap` занимает в памяти совершенно другое место, чем переменная `x` в функции `main`. Вы можете проверить это сами, распечатав адреса этих переменных:

```
#include <stdio.h>

void swap(int x, int y) {
```

```
  x ^= y;
  y ^= x;
  x ^= y;
  printf("swap: &x=%p x=%d &y=%p y=%d\n", &x, x, &y, y);
}

int main() {
  int x = 5, y = 7;
  swap(x, y);
  printf("main: &x=%p x=%d &y=%p y=%d\n", &x, x, &y, y);
  return 0;
}

Запуск программы:
swap: &x=0x7ffeef406afc x=7 &y=0x7ffeef406af8 y=5
main: &x=0x7ffeef406b18 x=5 &y=0x7ffeef406b14 y=7
```

Для печати адресов мы воспользовались `спецификатором формата %p`[6]

Чтобы разрешить изменять функцией `swap` переменные `x` и `y`, мы должны это явным образом указать: передать в `swap` не сами переменные, а их адреса. Только тогда функция `swap` будет что-то в состоянии сделать с переменными, находящимися в `main`

```
#include <stdio.h>

void swap(int *x, int *y) {
    *x ^= *y;
    *y ^= *x;
    *x ^= *y;
    printf("swap: &x=%p x=%d &y=%p y=%d\n", x, *x, y, *y);
}

int main() {
    int x = 5, y = 7;
    swap(&x, &y);
    printf("main: &x=%p x=%d &y=%p y=%d\n", &x, x, &y, y);
    return 0;
}

Запуск программы:
```

[6]Кстати, мы убедились, что адреса, хранящиеся в указателях, могут принимать довольно большие значения и точно не поместятся ни в `int`, ни в `unsigned`. А вот в `unsigned long long` они поместятся.

```
swap: &x=0x7ffeec169b18 x=7 &y=0x7ffeec169b14 y=5
main: &x=0x7ffeec169b18 x=7 &y=0x7ffeec169b14 y=5
```

Теперь всё работает, так как надо. Конечно, функция `swap` теперь менее красивая, нам приходится каждый раз использовать операцию `взятие значения по адресу`, которая обозначается `*`. Зато мы решили поставленную задачу.

> Единственным способом изменить значение локальной переменной при передаче её в функцию — передать её адрес и принять его как указатель в вызванной функции.

Интересно, что `Си` — один из немногих языков, который позволяет при вызове функции легко определить, может ли изменить функция свои аргументы. Например, по вызову функции `swap` в `main` мы видим знаки `&` перед переменными и понимаем, что значения этих переменных после вызова функции `могут` поменяться.

Для тех, кто знаком с языком `Pascal` вызов функции с указателями в качестве параметров может напомнить вызов процедур с параметрами, помеченными ключевым словом `VAR`. На самом деле всё так в `Pascal` и происходит — в процедуру передаётся адрес аргумента. Правда, `Pascal` скрывает от пользователя этот факт, а в `Си` всё происходит на наших глазах, что может кому-то показаться неудобным. В языке `Go`, появившемся в 2010-х годах, передача аргументов в функции происходит точь-точь, как в `Си`, и внутри функций там приходится пользоваться операцией взятия значения по указателю `*`.

2.12.5 Указатели и структуры.

Структуры — тоже объекты языка и тоже имеют адреса, а значит к ним тоже применимо понятие указателя.

После объявления структуры `car` (страница 54) можно объявлять и указатели на объекты такого типа и брать адреса этих объектов:

```
car a, *pa = &a;
```

Объявив одиночный автомобиль `a`, мы выдели память под хранение всех полей структуры и эта память обязана располагаться рядом и в заданной описанием последовательности. Указатель на автомобиль `pa`, который инициализировали адресом автомобиля `a`.

А как обращаться к полю структуры, представленной в виде указателя? Будет ли правильным, если мы напишем `*pa.velocity = 123.4;`? Нет. Приоритет операции квалификации поля `.` выше, чем операции взятия значения по адресу `*`, поэтому порядок исполнения операций будет следующим: `*(pa.velocity) = 123.4;`, а это точно не то, что нам нужно. Можно писать так: `(*pa).velocity = 123.4;`, это верно, но не кажется ли это вам слишком громоздким?

Для обращения к элементу структуры по указателю на структуру применяется операция `->`, например `pa->velocity = 123.4;`

2.12.6 Указатели и массивы. Арифметика указателей.

Настало время подробнее понять, что происходит, когда мы пишем, например, `int a[100];`. Мы называли такие массивы *простыми*. Простота такого массива заключается в том, что под него выделяется память на момент его создания. Оказывается, имя массива (в данном случае `a`) не что иное, как указатель на выделенную системой память. Позвольте, но если `a` — указатель, то как с ним работать? У нас же имеется только операция `*` для обращения к той памяти, на которую указатель показывает. В обращении к элементам массива нам помогает *адресная арифметика*.

Предположим, что начальным адресом нашего массива `a` будет `10000`. Тогда, обращение `*a` разрешено и это будет нулевым элементом массива. К нулевому элементу массива можно обратиться и как `a[0]`. Давайте запишем первое выражение немного по-другому: `*(a+0)`. Тогда `a[0]` есть синоним к `*(a+0)`. Продолжаем дальше. Память под массивы гарантированно выделяется *непрерывным куском*, поэтому элемент под номером `1`, `a[1]` будет находиться по адресу `10004`, если размер одного элемента типа `int` равен четырём. `a[1]` есть синоним к `*(a+1)`, `a[i]` — к `*(a+i)`. То есть если арифметика становится не совсем обычной: к указателю, равному `10000` прибавили единицу и он стал равен `10004`. Для всех ли типов данных это верно? Для всех с одной поправкой: степень увеличения адреса зависит от размера элемента массива.

Указатели допускают следующие операции:

сложение указателя с целым `ptr + i`. Результат показывает на элемент того же типа, отстоящий на `i` элементов дальше `ptr`.

вычитание из указателя целого `ptr - i`. Результат показывает на элемент того же типа, находящийся за `i` элементов перед `ptr`.

вычитание указателей одного типа `ptr2 - ptr1`. Результат — количество элементов между адресами.

У нас ещё будет возможность попрактиковаться в арифметике указателей, когда мы будем решать конкретные задачи.

Можно ли передавать массивы в функции? Мы этого ещё не делали. Да, конечно! Для совсем простаков можно описывать аргументы функций как массивы:

```
int add_elems(int array[100]) {
   int i, sum = 0;
```

```
   for (i = 0; i < 100; i++) {
      sum += array[i];
   }
   return sum;
}
```

Почему я сказал «для простаков»? Потому, что в Си массив как целое передать в функцию невозможно. Если понимать, что имя массива — просто указатель на нечто, то вполне достаточно передать это имя в функцию. Учтите, после того, как массив, точнее, указатель на него, переданы в функцию, никто не способен определить его размер. Пока мы находимся в области видимости точки создания массива, количество элементов массива доступно.

```
#include <stdio.h>

void bar(int b[100]) {
   int sizebar = sizeof b / sizeof b[0];
   printf("sizebar = %d\n", sizebar);
}

void func() {
   int arr[100];
   int sizearr = sizeof arr / sizeof arr[0];
   printf("sizearr = %d\n", sizearr);
   bar(arr);
}

int main() {
   func();
}
```

Исполнение этой программы может кого-то удивить:

```
sizearr = 100
sizebar = 2
```

```
int add_array(int *arr}
```

Не очень наивный человек уже понимает, что в функцию передан указатель на массив, он занимает (на данной вычислительной системе) 8 байт, размер типа `int` равен четырём байтам, отсюда `sizebar = 2`.

Это, конечно, похоже на обман. Сделано это очень давно, в те времена, когда вычислительные программы писались на языке FORTRAN и решение разрешить объявления аргументов функции в виде массива было сделано для

того, чтобы не отпугнуть тех, кто хочет мигрировать с FORTRAN на Си. Но мы-то с вами умные и всё понимаем. Знать, что передан именно массив, а не указатель на отдельную переменную иногда полезно. Однако (увы!) число в квадратных скобках, которое, вроде бы, должно показать нам количество элементов массива, просто игнорируется[7]. Следующие три объявления функции эквивалентны:

```
int add_elems(int arr[10000]);
int add_elems(int arr[]);
int add_elems(int *arr);
```

Операция `sizeof arr` в любом случае возвратит размер указателя, а не массива. Ну а раз так, у нас нет возможности узнать, сколько в массиве элементов. Обычно в таких случаях передают размер массива в следующем за именем массива аргументе:

```
int add_elems(int *array, int n) {
    int i, sum = 0;
    for (i = 0; i < n; i++) {
        sum += array[i];
    }
    return sum;
}
```

Теперь у нас есть универсальная функция, которая готова вычислить сумму элементов любого целочисленного массива любой длины. В классическом Pascal такую функцию написать было невозможно (удивительно, но создатель языка не подумал об этом). Хотите найти сумму элементов массива длиной 10 — одна функция, массива длиной 20 — другая функция. Работа с матрицами и векторами, необходимая для вычислительной математики, была, таким образом саботирована. Только в конце 1980-х в Pascal появились *открытые массивы*, что, по сути, и есть замаскированные указатели. Но уже было поздно и математики, которые хотели уйти с FORTRAN, ушли на Си.

2.12.7 Многомерные массивы

Простые многомерные массивы практически не отличаются от простых одномерных массивов:

```
#include <stdio.h>

int sum_matrix(int m[3][3]) {
```

[7] Это не так для многомерных массивов, но об этом — позже

```
  int sum = 0;
  int i,j;
  for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
      sum += m[i][j];
    }
  }
  printf("sizeof m=%d\n", sizeof m);
  return sum;
}

int main() {
  int matrix[3][3];
  int i,j;
  for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
      scanf("%d", &matrix[i][j]);
    }
  }
  int s = sum_matrix(matrix);
  printf("sizeof matrix=%d sum=%d\n", sizeof matrix, s);
}
```

Попытка откомпилировать и исполнить программу ещё раз убеждает нас, что в функцию передаётся только указатель (`sizeof m = 8`):

```
1 2 3
4 5 6
7 8 9
sizeof m=8
sizeof matrix=36 sum=45
```

Но ведь для того, чтобы пользоваться двумерными массивами, необходимы два индекса. Мы не можем передать просто указатель

```
int sum_matrix(int *m) { ...
```

так как тогда невозможна *двойная индексация* `m[i][j]`.

Чтобы сохранить «двумерность» нам можно объявить функцию так:

```
int sum_matrix(int m[][3]) {
```

Это странное объявление говорит, что при адресации мы должны использовать двойную индексацию и что каждая строка состоит ровно из трёх элементов. Это позволяет вычислять адрес любого из элементов массива `m[i][j]` по формуле `m+i*3+j`.

Фактически компилятор превращает нашу функцию в

```
int sum_matrix(int *m) {
    int sum = 0;
    int i,j;
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            sum += m[i*3+j];
        }
    }
    printf("sizeof m=%d\n", sizeof m);
    return sum;
}
```

Простые многомерные массивы хранятся в Си в *линеаризованном виде*, строка за строкой. В функции передаётся адрес занятой памяти. Разрешается (только в аргументах функции!) оставить пустые `[]` для самого левого измерения массива, например, вместо `double x[5][5][5]` писать `double x[][5][5]`. Заметьте, что принимаемые в функции двумерные и многомерные массивы должны содержать константные выражения в каждой из квадратных скобок.

2.12.8 Куча

Те массивы, которые мы объявляли до этого имели время жизни или равное времени жизни всей программы (статические и глобальные массивы), или время жизни, равное времени жизни блока, а котором они описаны (автоматические или стековые массивы). А можно ли в какой-либо функции *создать* массив такой, что его не будет при входе в функцию и он сохранится при выходе из неё? Статические массивы не подходят — они существую и *до* входа в функцию. Автоматические массивы тоже на подходят — они уничтожаются *после* выхода из функции. Нам на помощь приходят *динамические* массивы, которые создаются в области памяти под смешным названием *куча* или по английски *heap*.

Запросить память из кучи (ещё говорят «заказать память») можно, используя вызовы нескольких функций. Разумеется, все функции должны возвращать адреса кусков памяти, в которых будут размещаться, например, массивы. Следовательно, возвращаемое значение функций должно быть указателем. Но на что? На всё. В Си есть специальный тип для указателя, которые готов представить любой адрес — `void *`. Мы говорили, что `int *` может оказаться массивом `int`, то есть целых, но что можно сказать `void *`? Нет, массивов такого типа не бывает. Нельзя написать:

```
int f(void *arg) {
   return arg[0];
}
```

Зато этот указатель присвоить указателю нужного типа и далее работать с тем, как обычно.

```
int f(void *arg) {
  int *a = arg;
  return a;
}
```

Хотя Си и разрешает такие присвоения, следующий за ним язык — C++ уже так делать не позволяет и там нужно писать так:

```
int f(void *arg) {
  int *a = (int *)arg;
  return a;
}
```

Так как мы вскоре начнём знакомиться и с элементами C++, рекомендую привыкать к такому поведению.

Не забыли ли мы про функции? Для их использования нужно включить заголовочный файл <stdlib.h>, не забывайте про него. Вот первая из функций: malloc.

```
  int *a = (int *)malloc(n * sizeof(int));
```

У неё один аргумент — количество байтов, которые мы просим у системы. Если система готова нам их предоставить — результат возвращается функцией и его можно присвоить какому-либо указателю, что мы и сделали. Теперь указатель `int *a` содержит адрес фрагмента памяти, в котором можно хранить не менее `n` элементов типа `int`, то есть по сути — массив. В таком случае говорят: заказать память под массив или заказать *динамический* массив.

Теперь тот, кто получил этот указатель, несёт ответственность за жизнь этого указателя — пока этот указатель есть единственный способ обратиться к памяти из кучи. Потеряем указатель — потеряем эту память. Возникнет ситуация, известная, как *утечка памяти* или *memory leak*. Она крайне неприятна и в ряде случаев приводит к ситуациям, когда программа просто становится неработоспособной. Потерять память — проще простого:

```
int foo(int n) {
   int *bar = (int *)malloc(n * sizeof (int));
   return 0;
}
```

Так как переменная `bar`, содержащая адрес свежевыделенного системой куска памяти, после достижения закрывающей фигурной скобки исчезает, вся выделенная память внезапно становится недоступной. То есть она как бы есть, но её как бы и нет.

> Память, выделенная `malloc`, неинициализирована. Чтобы получить инициализированный нулями кусок памяти, используйте `calloc`. Он требует два аргумента: количество заказываемых элементов и размер единичного элемента: `int *s = (int *)calloc(n, sizeof(int));`

Освободить память, то есть вернуть её в систему, можно функцией `free`.

> В корректных программах каждый заказ памяти по `malloc/calloc/realloc` должен иметь парный вызов `free`.

У автора в его обширной практике был один памятный случай, связанный с утечкой памяти. Был сдан в опытную эксплуатацию комплекс программ по мониторингу и управлению одним из крупных объектов энергетического комплекса. Одна из программ на `C++` предназначалась для сбора, обработки информации и распределению её по базам данных и рабочим станциям, и она должна была работать круглосуточно. Всё шло хорошо, но через примерно полгода эксплуатации заметили, что сервер, на котором она была запущена, стал работать медленнее. Анализ состояния сервера обнаружил, что данная программа занимает в памяти более 40 мегабайт, что при общем количестве памяти на сервере в 16 мегабайт и привело к существенному замедлению работы (на вопрос: *а как программа может занимать больше памяти, чем имеется на компьютере?* отвечу, что подробный ответ вы узнаете на втором курсе при изучении операционных систем). Анализ кода программы показал, что в одном из параллельно исполняющихся потоков программы, который активировался каждую минуту, не освобождалась память, общим размером 32 байта. За шесть месяцев это привело к утечке почти 30 мегабайт памяти.

2.12.9 Куча: двумерные массивы

2.13 Стандартный ввод/вывод

2.13.1 Потоки ввода/вывода

Работа с файлами, содержащими текст, в `Си` мало чем отличается от работы с клавиатурой и экраном — все старые навыки оказываются полезными. Когда мы пишем `printf("Hello\n");` мы какой-то текст (`"Hello\n"`) хотим куда-то вывести. Куда? По-умолчанию на экран. На самом деле, экран, как и многие другие похожие сущности (жёсткий диск, DVD-диск, принтер) с точки зрения `Си` являются *файлами* и к ним всем применимы одни и те же операции.

В `<stdio.h>`, как мы знаем, располагаются прототипы таких функций, как `printf, scanf, getchar`. Там, наряду с другими, имеется константа `EOF`. Что нового: там имеется описание типа данных `FILE`. Этот тип данных поможет

нам читать/писать информацию, расположенную в файлах. Мы только обратили внимание на то, что экран (давайте называть это лучше *стандартный вывод*) тоже является файлом, и клавиатура (с этого момента мы будем называть её *стандартный ввод*) — это тоже файл. `FILE` — системная структура данных, которую мы будем передавать далее в функции работы с файлами, а структуры, как мы знаем, передаются по значению, то есть копируются. Именно по этому все функции работы с файлами в качестве аргументов используют не саму структуру `FILE`, а указатели на объекты такого типа есть `FILE *`. Три таких указателя уже имеются, они объявлены в `<stdio.h>` и ими можно пользоваться. Это следующие глобальные переменные:

```
FILE *stdin;
FILE *stdout;
FILE *stderr;
```

Первая структура обеспечивает доступ к стандартному вводу. Следующие две строки эквивалентны:

```
scanf("%d", &n);
fscanf(stdin, "%d", &n);
```

Вторая структура, как нетрудно догадаться, обеспечивает доступ к стандартному выводу и функция `printf` тоже имеет своего двойника:

```
printf("Hello, I'm robot number %d\n", robot_number);
fprintf(stdout, "Hello, I'm robot number %d\n", robot_number);
```

Третья структура обеспечивает доступ к `стандартному выводу ошибок` и тоже вначале связана с экраном. Как вы потом убедитесь, каждый из *стандартных* вводов и выводов может быть `перенаправлен`, но об этом мы сейчас говорить не будем.

Мы с вами научились выводить информацию в любой файл и вводить информацию из любого файла. Разве? Мы же использовали только стандартные ввод и стандартный вывод. Если вместо `stdin` или `stdout` мы подставим свои указатели на `FILE`, то своей цели добьёмся. Осталось только понять, как такие указатели можно получить.

2.13.2 Файловый ввод/вывод

Используем функцию `fopen`. Она принимает в своих аргументах имя файла и режимы его открывания (только читать, только писать, читать и писать и тому подобное), а возвращает необходимый для операций с файлами указатель на `FILE`:

```
FILE *fpr = fopen("input.txt", "r"); // "r" --- только для чтения
FILE *fpw = fopen("output.txt", "w"); // "w" --- для записи
FILE *fpa = fopen("append.txt", "a"); // "a" --- для добавления
```

К строке с режимами можно добавить знак +: `"r+"` или `"w+"`. Это означает, что мы после чтения файла хотим туда что-то записать или после записи файла что-то оттуда прочитать. Для этого можно файл *перемотать на начало:* `rewind(fp);`

Внимание: если вы использовали режим `"w"`, то существующий до этого файл с тем же именем вначале будет обнулён.

Функция `fopen` возвращает указатель на структуру `FILE`, и если файл по каким-то причинам не может быть открыт, например, если вы его открываете на чтение, а он не существует, то функция возвращает `NULL`.

После окончания работы с файлом, открытым с помощью `fopen`, его обязательно надо `закрыть`

```
fclose(fpr);
fclose(fpw);
fclose(fpa);
```

А что будет, если мы попытаемся закрыть не открытый нами файл, такой, как `stdin` или `stdout`? Всё, что угодно. Скорее всего, произойдёт то, что мы могли бы ожидать — перестали вводиться данные со стандартного ввода или перестали поступать данные на экран по `printf`. Но может произойти и аварийное завершение программы. Так что не делайте этого.

А вот что произойдёт, если мы попытаемся закрыть файл, указатель на который равен `NULL`? В `Visual Studio` программа аварийно завершится. В других компиляторах всё может произойти тихо и мирно — функция `fclose(fp)` при `fp==NULL` игнорирует некорректный указатель.

А если мы попытаемся закрыть уже однажды закрытый файл? Ничего хорошего. Поэтому типичный код при закрытии файла похож на следующий:

```
if (fp != NULL) {
fclose(fp);
fp = NULL;
}
```

Этот код стабильно работает во всех случаях и предпочтительнее использовать именно подобный шаблон.

2.14 Опять строки

Мы можем возвратиться к строкам после того, как изучили указатели, так как строки на указателях и основаны. Вы же помните, что строки в `Си` — дан-

ные чрезвычайно низкого уровня? Так вот низкий уровень строк служит основой довольно высокой эффективности алгоритмов работы с ними. Основным источником неэффективности строк является операция нахождения их длины. В ряде алгоритмов можно уменьшить влияние этой неэффективности, в ряде — этого сделать не удаётся. Впрочем, современные компиляторы знают про строки всё и часто могут оптимизировать операции с ними. В стандартной библиотеке языка Си имеется заголовочный файл, посвящённый операциям над строками и мы сейчас рассмотрим некоторый операции из него на примере.

```
#include <string.h>

int main() {
   const char *s1 = "Hello";
   int l1 = strlen(s1);

}
```

Перечислим некоторые из функций, имеющихся в заголовочном файле string.h

```
// Заказать память для хранения нужного количества
// символов и скопировать туда источник
char *s = strdup("Hello");
// Не забудьте сделать free после использования!
int len = strlen(s); //  Длина строки.  Сейчас 5.
char *t = malloc(1000);
strcpy(t,s); // Скопировать "Hello" в первые 5 байтов
       // t и добавить к конку \0
strcat(t, " "); // Теперь в t лежит "Hello "
strcat(t, "world!"); // "Hello world!"
strncat(t, " very long string", 1000);
// третий аргумент --- максимальная длина приёмника
char q[10];
strncpy(q,"Try to copy very-long-string", 10);
// Будет скопировано первые 10 символов.
// Но нулевого байта может не оказаться!!!
int len = strlen(q); // Что угодно >= 10! Осторожно!
char p[10];
strcpy(p,"Try to copy very-long-string"); //  Катастрофа!!!
```

Функция strcmp — одна из полезнейших и похоже, что она применяется чуть ли не чаще других в практических применениях. Её прототип такой:

```
int strcmp(const char *src, const char *dst);
```

Хотя строки, которые подаются как аргументы, могут быть произвольной длины, сама функция довольно ленива и может выдавать ответ исключительно быстро. А вообще, что значит сравнить строки? Является ли более длинная строка большей? Нет. Упорядочиваются функции **лексикографически**, то есть так, как в словаре располагаются словарные статьи. Ну а раз так, то алгоритм сравнения строк прост: символы двух строк сравниваются попарно до тех пор, пока они совпадают. После чего возвращается разность кодов несовпавших символов. Например, при сравнении `"cadabra"` и `"cadence"` функция сравнит сначала первые буквы (это `'c'`), затем вторые (`'a'`), третьи (`'d'`) и, обнаружив, что буква 'a' меньше буквы 'e', вернёт их разность со знаком, то есть -4. Строки могут совпасть тогда и только тогда, когда они имеют одинаковую длину и одинаковое содержимое. Тогда функция (внимательно!) возвратит ноль. Если первая строка больше второй, то результатом будет что-то положительное.

Функция `strcmp` сравнения двух строк возвращает значение 0, если строки равны. Поэтому типичный код такой:

```
if (strcmp(op, "div") == 0) ...
```

Ещё одна функция, без которой обойтись сложно — функция `strlen`, возвращающая длину строки-аргумента.

Функция `strlen(const char *s)` возвращает длину строки без завершающего нулевого байта, то есть `strlen("Hello")` = 5.

Нельзя сказать, что функция `strlen` работает уж очень быстро. Время её исполнения прямо пропорционально длине строки, так что для строки длиной в миллиард байт она будет исполняться уже доли секунды (напоминаю, что современный, 2018-го года компьютер без труда перемалывает миллиарды операций в секунду).

2.14.1 Пишем функции работы со строками

Легко ли писать функции работы со строками? Вполне. Давайте попробуем сделать это на примере функции `strlen`. Алгоритм прост: мы бежим по строке до тех пор, пока не встретим нулевой байт.

```
int mystrlen(const char *s) {
   int i = 0;
   while (s[i] != 0) {
      i++;
   }
   return i;
}
```

Весьма просто и к тому же достаточно эффективно. Впрочем, я предостерегаю вас от того, чтобы самостоятельно реализовывать эту функцию. Проблема не в том, что вы не правильно это сделаете, а в том, что современные компиляторы знают, что эта функция используется очень часто и *генерируют* код, который работает быстрее, чем то, что мы написали. Впрочем, это уже тема второго семестра.

Функцию `strcpy` тоже можно реализовать самостоятельно.

Достаточно примитивный, но работающий вариант состоит в том, что мы сначала находим длину строки, которую копируем и в цикле производим копирование.

```
void mystrcpy(char *d, const char *s) {
   int i, len = mystrlen(s);
   for (i = 0; i <= len; i++) {
      d[i] = s[i];
   }
}
```

Обратили внимание на то, что мы написали непривычно: `i <= len`? Не ошибка ли это? Отнюдь. Сначала мы скопировали `len` символов самой строки, а затем поместили нулевой байт для её завершения.

> Внимание! При копировании строк в обязаны быть уверены, что указатель `char *d` указывает на кусок реальной памяти не менее `len + 1` байтов длиной. Её можно получить, например, вызовом `calloc/malloc` или объявлением вида `char dest[100];`

Не кажется ли вам, что наша функция неэффективна? В самом деле, для получения длины строки `s` мы пробежали до её конца и при копировании бежим по ней ещё раз. Давайте воспользуемся циклом с завершением по нулевому байту.

```
void mystrcpy(char *d, const char *s) {
   int i = 0;
   while (s[i] != 0) {
      d[i] = s[i];
      i++;
   }
   d[i] = 0;
}
```

Вроде бы сложность функции уменьшилась. Можно остановиться? А давайте ещё попробуем улучшить код. Возможное место улучшения — цикл. В цикле мы производим две операции — сравнения очередного символа правой

строки с нулём, и если он не равен нулю, копируем. Мы верим, что наш компилятор сможет использовать операнд сравнения `s[i]` для присваивания этого же значения в `d[i]`, но можем это указать явно:

```
void mystrcpy(char *d, const char *s) {
   int i = 0;
   while ((d[i] = s[i]) != 0) {
      i++;
   }
}
```

Мы оказались достаточно ловкими: сначала присвоили значения `d[i] = s[i]`, а потом воспользовались тем, что любая операция присваивания в Си имеет значение, равное её левому операнду и сравнили результат с нулём. Если мы достигли завершающего символа, равного нулю, то присваивание мы всё равно произвели и только после этого покинули цикл. Это уже неплохо и смотрится более эффективно.

Напоследок приведём пример самой изящной реализации функции `mystrcpy`:

```
void mystrcpy(char *d, const char *s) {
   while (*d++ = *s++)
      ;
}
```

Попробуйте в ней разобраться и убедиться в том, что всё работает корректно.

2.15 Небольшая практическая задача

Задача 2.6. Во входном файле `input.txt` содержится список персонала — строки, каждая из которых состоит из четырёх полей: фамилия, имя, отчество и заработная плата. Все поля разделены пробелами. Требуется вывести в файл `output.txt` тот же самый список, отсортированный по убыванию заработной платы.

Перед решением задачи мы должны понять, на какие части мы должны её разбить, чтобы каждая часть была не особенно трудной и чтобы из решения частей можно было собрать общее решение. В денном случае разбиение на подзадачи не кажется особенно сложным:

1. Считать весь файл в массив.
2. Отсортировать массив в порядке убывания зарплаты.
3. Вывести массив в выходной файл.

При решении первой подзадачи сразу же возникает вопрос: `массив чего?` и на него быстро же находится ответ: массив структур. Каждую персону мы будем хранить в виде одной структуры.

Следующие вопросы: умеем ли мы работать с информацией, хранящейся в файлах? Какие операции с файлами мы можем производить? Как обнаружить, что в файле записей больше нет? К счастью, с файлами работать мы умеем, это было в разделе 2.13. Это же умение позволяет решить нам третью подзадачу.

Вторая подзадача кажется достаточно сложной, так как она имеет отношение к алгоритмам, что уже является темой следующей части. Однако в нашем случае не требуется тщательный выбор алгоритма, так как, к нашему удовольствию, в стандартной библиотеке `Си` уже имеется реализация алгоритма сортировки `qsort`, которой мы и воспользуемся.

Итак, в общих чертах нам понятно, что в состоянии справиться с каждой из трёх подзадач, так что давайте приступим к ним.

2.15.1 Первая подзадача: считать весь файл в массив

В условии задачи не сказано, сколько элементов будет в массиве. Мы можем сказать себе: количество людей на предприятии не может быть уж очень велико, давайте поставим ограничение на длину массива, например, в 10000 и решим задачу с ограничениями. Хорошо, на первом этапе реализации так можно сделать и если далее потребуется, изменить реализацию так, чтобы это ограничение снять в дальнейшем.

Теперь нам требуется уточнить представление структуры. Будем ли мы хранить каждый элемент в виде массива символов определённой длины? Если да, то сколько символов потребуется для хранения каждого элемента структуры — фамилии, имени и отчества? Хватит ли, скажем, двадцати символов? В первой *итерация* решения задачи давайте позволим такое с нашим внутренним обязательством уточнить решение в дальнейшем.

> При решении большой задачи неплохо создать корректно работающий черновик, который корректно принимает большинство из возможных входных данных и который в дальнейшем будет уточняться для более полного решения.

Итак, пока наша структура (давайте назовём её `person`) имеет следующий вид:

```
typedef struct person_s {
   char surname[20]; // Фамилия
   char name[20];      // Имя
   char secname[20]; // Отчество
   double salary;        // Зарплата
} person;
```

Давайте реализуем первую подзадачу в виде функции, которая на вход получает имя входного файла, а на выходе возвращает указатель на массив из структур `person`. Такая организация позволит нам и разбить задачу на относительно независимые подзадачи, и достаточно легко вносить изменения в организацию данных. К сожалению, `Си` не позволяет вернуть из функции по оператору `return` сразу несколько значений. Поэтому прототипом функции сделаем `person *prepare(const char *filename, int *num_of_persons);`

Первым параметром будет передано имя файла, это указатель, квалификатор `const` показывает, что мы не собирается ничего менять по его содержимому. Второй параметр — возвращаемое функцией значение, следовательно, он должен быть указателем. Можно ли было сделать изящнее? Да, конечно, чуть попозже мы попробуем это сделать.

```
person *prepare(const char *filename, int *num_of_persons) {
  FILE *in = fopen(filename, "r");
  if (in == NULL) {
     return NULL; // Невозможно работать --- нет файла
  }
  const int MAX_SIZE = 10000;
  person *ret = malloc(MAX_SIZE * sizeof(person)); // Этот указатель
     // мы возвратим в конце работы функции.
  int num;
  char surname[100], name[100], secname[100];
  double salary;
  for (num = 0; num < MAX_SIZE &&
    fscanf(in, "%s %s %s %lf\n",  surname, name, secname, &salary) == 4;
    num++) {
     person *p = ret+num;
     strncpy(p->surname,20, surname);
     strncpy(p->name, 20, name);
     strncpy(p->secname, 20, secname);
     p->salary = salary;
  }
  fclose(in);
  *num_of_persons = num;
  ret = realloc(ret, num * sizeof(person));
  return ret;
}
```

Можно ли сказать, что написанная нами функция работает корректно во всех возможных случаях? Нет, конечно. Она, конечно, не такая уж и наивная, но она никак не известит того, кто её вызвал, о причинах невозможности открыть файл и она будет работать совсем плохо если окажется, что в исходном файле кто-то по ошибке или по злому умыслу разместил фамилию,

имя или отчество длиной более 100 символов. Если мы поняли, что функция делает что-то не так в ряде случаев, у нас имеется возможность изменить её внутренности, не затрагивая её пользователя.

2.15.2 Вторая подзадача: отсортировать массив

Первая подзадача дала нам исходные для второй подзадачи данные: массив структур и его размер. Для сортировки массива воспользуемся функцией `qsort`, описанной в `stdlib.h`.

Эта функция имеет четыре аргумента. Первый аргумент — указатель на сортируемые данные. Второй аргумент — количество сортируемых элементов. Третий — размер сортируемого элемента в байтах. Это всё просто и привычно. А вот четвёртый аргумент достаточно необычен. Это — указатель. Но указатель не на какие-либо данные, а на функцию. Перед тем, как пользоваться функцией `qsort`, требуется написать функцию, которая поможет сортировать элементы или так называемую *функцию сравнения*. Эта функция должна принимать ровно два аргумента-указателя. А вот на что они должны указывать? Разумеется, они должны содержать адреса сравниваемых элементов. Но вот беда, до того, как мы начали писать нашу программу, функция `qsort` не имела ни малейшего понятия о типе данных `person` и о указателях на такой тип данных. Поэтому используется так называемый `универсальный указатель`, который может хранить в себе любой доступный программе адрес. Его тип — `void *`, то есть указатель на ничто (что, скорее, лучше читать, как указатель на нечто). Указатели этого типа можно свободно присваивать любым другим указателям и наоборот.

Функция, сравнивающая элементы, должна возвращать значение, которое трактуется следующим образом: если это значение отрицательно, то первый элемент должен стоять в отсортированном массиве перед вторым; если нуль, то элементы равны и порядок элементов, возможно, должен быть сохранён (забегая вперёд скажем, что сортировка, сохраняющая порядок элементов в массиве, называется `устойчивой`, подробнее об этом в **??**)

Итак, вот наша функция сравнения элементов:

```
int person_compare(const void *l, const void *r) {
   const person *pl = l;
   const person *pr = r;
   if (pl->salary > pr->salary) return -1;
   return pl->salary < pr->salary;
}
```

Чтобы получить доступ к полю структуры `person` под именем `salary`, мы должны преобразовать универсальный указатель, который был передан в функцию, в указатель нужного типа. Как мы уже знаем, для такого преобразования вполне достаточно скопировать его в созданный нами указатель на

структуру `person`. Если зарплата первого сотрудника больше зарплаты второго, то первый должен быть в массиве *перед* вторым, а это означает, что функция сравнения должна вернуть отрицательное значение. Если зарплаты окажутся равны, то значением операции сравнения в последней строке функции будет ложь, то есть 0. А если не равны, то останется один вариант — зарплата первого меньше зарплаты второго (ведь мы уже покинули функцию раньше, если первый был больше второго), результатом операции сравнения будет 1, чего нам и хотелось бы.

Итак, мы готовы написать и решение второй подзадачи (вместе с вызовом функции, решающей первую):

```
int num_of_persons;
person *persons = prepare("input.txt", &num_of_persons);
qsort(persons, num_of_persons, sizeof (person), person_compare);
```

На выходе мы получили отсортированный массив персон и наступает время решать третью подзадачу:

2.15.3 Третья подзадача: вывести отсортированный массив

Здесь, как будто, всё просто.

```
FILE *out = fopen("output.txt", "w");
if (out != NULL) {
  int i;
  for (i = 0; i < num_of_persons; i++) {
    fprintf(out, "%s %s %s %.2f\n", persons[i].surname,
    persons[i].name, persons[i].surname, persons[i].salary);
  }
  fclose(out);
} else {
  printf("Всё плохо, шеф\n");
}
```

Как будто всё? Нет. Теперь полагается почистить рабочее место, то есть вернуть всю заказанную нами у системы память. Ожидаю вопроса: *Зачем это? Ведь программа сейчас завершится сама и память вернётся в систему.* Отвечаю: если написанный вами алгоритм не будет освобождать всю заказанную им для своих целей память, то этот алгоритм будет невозможно применять в программе несколько раз, будет происходить *утечка памяти.*

> Хорошие алгоритмы должны быть подобны чёрному ящику: они должны принимать некий вход и формировать некий выход. Всё изменение содержания окружающей среды будет восприниматься как побочный эффект работы алгоритма, что чаще всего недопустимо.

Последний штрих:

```
free(persons);
```

Итак, полный вариант нашей программы:

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

typedef struct person_s {
  char surname[20]; // Фамилия
  char name[20];      // Имя
  char secname[20]; // Отчество
  double salary;        // Зарплата
} person;

int person_compare(const void *l, const void *r) {
  const person *pl = l;
  const person *pr = r;
  if (pl->salary > pr->salary) return -1;
  return pl->salary < pr->salary;
}

person *prepare(const char *filename, int *num_of_persons) {
  FILE *in = fopen(filename, "r");
  if (in == NULL) {
    return NULL; // Невозможно работать --- нет файла
  }
  const int MAX_SIZE = 10000;
  person *ret = malloc(MAX_SIZE * sizeof(person)); // Этот указатель
  // мы возвратим в конце работы функции.
  int num;
  char surname[100], name[100], secname[100];
  double salary;
  for (num = 0; num < MAX_SIZE &&
   fscanf(in, "%s %s %s %lf\n",  surname, name, secname, &salary) == 4;
   num++) {
    person *p = ret+num;
    strncpy(p->surname, surname, sizeof p->surname);
    strncpy(p->name, name, sizeof p->name);
    strncpy(p->secname, secname, sizeof p->secname);
    p->salary = salary;
```

```
  }
  fclose(in);
  *num_of_persons = num;
  ret = realloc(ret, num * sizeof(person));
  return ret;
}

int main() {
   int num_of_persons;
  person *persons = prepare("input.txt", &num_of_persons);
  if (persons == NULL) {
     printf("Не удалось открыть входной файл");
     return 1;
  }
  qsort(persons, num_of_persons, sizeof (person), person_compare);
  FILE *out = fopen("output.txt", "w");
  if (out != NULL) {
    int i;
    for (i = 0; i < num_of_persons; i++) {
      fprintf(out, "%s %s %s %.2f\n", persons[i].surname,
         persons[i].name, persons[i].surname, persons[i].salary);
    }
    fclose(out);
  } else {
    printf("Всё плохо, шеф\n");
  }
  free(persons);
  return 0;
}
```

Эту задачу, конечно, можно усовершенствовать многими способами. Например, не кажется ли вам, что мы слишком много тратим памяти на хранение всякой пустоты? Например, чтобы хранить имя `Иван` было бы достаточно пяти символов, между тем мы расходуем двадцать. Возможно, найдётся длинное имя, фамилия или отчество, которые будут длиннее 19 символов. Если мы зарезервируем под них 25 или 30 байтов, то для коротких имён потери будут ещё больше. Как хранить короткие имена?

3 Снова алгоритмы

3.1 Этапы решения задачи: Анализ. Декомпозиция. Синтез

Итак, мы вернулись к алгоритмам. При решении больших задач невозможно получить решение, не разбивая задачу на подзадачи. Подзадачи, в свою очередь, тоже могут делиться на подподзадачи и так до тех пор, пока мы не сможем сказать: я знаю, как решаются все подзадачи и я готов, решив их, собрать из результатов нечто целое. Фазы этого процесса имеют свои названия:

1. Перед началом разбиения мы производим *анализ* задачи, определяя, стоит ли вообще производить разбиение, и если стоит, то на какие компоненты. Хорошее разбиение даёт возможность потратить меньше времени на процесс решения.

2. После того, как мы определили, на какие подзадачи мы будем разбивать наши задачи, мы на время забываем о задаче в целом, произведя *декомпозицию* и ищем решения всех подзадач.

3. После того, как решения подзадач найдены, наступает этап *синтеза* — сборки из решений отдельных подзадач единого целого.

3.2 Абстракции

Как только появляются *объекты*, появляются *абстракции* — механизм разделения сложных объектов на более простые, без деталировки подробностей разделения.

Функциональная абстракция — разделение функций, *методов*, которые манипулируют с объектами с их реализацией.

Интерфейс абстракции — набор методов, характерных для данной абстракции.

Пример: абстракция массива

- **create** — Создать массив. Статический или динамический?

```
int a[100]; // Статический
int *b = calloc(100, sizeof(int)); // Динамический
int *c = new int[100]; // Динамический (это уже C++)
```

- **destroy** — Удалить массив. Статический или динамический?

```
free(b);
delete c; // можно и delete [] c, опять пример из C++
```

- **fetch** — Обратиться к элементу массива.

```
int q1 = a[i];
int q2 = b[i];
int q3 = c[i];
```

Для массива основная операция — это доступ к элементу. Она выглядит одинаково для всех представлений.

Абстракции мы будем использовать как некие *точки притяжения* — то есть как некоторые маячки, подзадачи, решение которых мы знаем.

3.3 Индукция и инвариант

Для создания хороших алгоритмов, в том числе и для для работы с массивами и, главное, доказательства их корректности, применяется несколько математических принципов. Один из наиболее плодотворных — принцип индукции. Как решить задачу «поиск минимума» в массиве, то есть найти элемент массива, значение которого минимально среди всех элементов массива? Под термином «найти элемент» можно понимать как нахождение индекса элемента, так и нахождение его значения.

Операция минимума обладает свойствами *коммутативности* (`min(a,b) = min(b,a)`) и *ассоциативности* (`min(a,(min(b,c)) = min(min(a,b),c)`). Это означает, что мы можем проводить эту операцию над элементами массива в любом порядке.

Второй математический принцип, полезный для создания алгоритмов — принцип *инвариантности*.

> *Инвариант* в программировании — утверждение о значениях переменных или их взаимоотношении, истинность которого не изменяется при исполнении алгоритма.

В данном случае в качестве инварианта выберем утверждение: *переменная* ***min*** *после шага* ***k*** *есть минимальное значение подмассива* ***[0..k]***.

Индукцию мы начинаем с *базы* — утверждения, что минимум подмножества из одного элемента есть сам элемент.

```
min = ar[0];
```

Следующий шаг: *индуктивный переход*. Если утверждение было истинно на шаге `k` и переход от шага `k` к шагу `k+1` не нарушил инвариант, то для любого `n` утверждение окажется истинным.

```
for (i = 1; i < n; i++) {
  if (a[i] < min) {
    min = a[i];
  }
}
```

Возможно, что подобные рассуждения могут показаться сложными и избыточными, но мы хотели показать, что можно совместить разработку алгоритмов с одновременным доказательством их корректности. К сожалению, большое количество разработчиков программ не утруждает себя доказательствами корректности созданного ими продукта. Конечно, много алгоритмов уже известно и реализовано, но без умения доказывать корректность своих действий невозможно достичь настоящего профессионализма, так же, как это невозможно и в математике.

А что дальше? А дальше — ещё 344 страницы текста, посвящённого различным абстракциям, алгоритмам и структурам данных. Моя книга опубликована в издательстве МаксПресс в 2019 году и с разрешения издательства она выложена по адресу Лекции по алгоритмам и структурам данных.

Книга продолжает развиваться, исправляются замеченные неточности, добавляются новые разделы. Последний вариант книги по алгоритмам.

Предметный указатель