Ловягин Никита Юрьевич
Алимова Ольга Викторовна

Практикум на ЭВМ

лабораторные работы и задачи

для студентов прикладных
математических специальностей

Санкт-Петербург
2017

Оглавление

Список лабораторных работ

Часть I. Введение

Глава 1. Основы создания программ на языке Си

§1.1. Предварительные сведения

Основные понятия

Прежде чем начинать изучение языка Си следует ввести ряд ключевых понятий. В практическом курсе они вводятся неформально.

Во-первых, следует уяснить, что компьютеры понимают т.н. **машинные команды** или **коды**, представляющие собой наборы некоторых чисел. Человеку работать с такими командами неудобно, поэтому существует буквенная запись данных команд. Наиболее близким к кодированию с помощью буквенного представления машинных команд является язык **ассемблера**. Машинные коды и язык ассемблера используют простейшие синтаксические конструкции и команды, это языки программирования низкого уровня. Создавать программы с их помощью можно, но, трудоемко в силу элементарности инструкций — для того, чтобы создать что-то сложное необходимо написать много инструкций, то есть длинную программу — а также неудобно в силу отсутствия синтаксических управляющих конструкций, позволяющих, в частности, представить структуру программы более наглядно, что является источником труднообнаружимых ошибок. Существуют и другие причины, в силу которых использовать машинные коды следует лишь там, где это действительно необходимо, например, при создании языков программирования и некоторых элементов операционных систем. В противоположность языкам программирования низкого уровня разработаны **языки программирования высокого уровня**, лишенные указанных недостатков. "Платой" за использование языков программирования высокого уровня является более медленно работающие программы, однако, строго говоря, это не всегда так. В частности язык программирования Си является инструментом создания одних из самых быстро работающих программ, кроме того, современные компиляторы позволяют *оптимизировать* исполняемый код, то есть делать его более быстрым, чем если бы перевод кода с языка высокого уровня в машинный производится буквально.

Во-вторых, даже программу, написанную на ассемблере, а тем более написанную на языке программирования высокого уровня (ЯВУ), следует перевести в машинный код, то есть из **исходного кода** получить **исполняемый файл**. Это осуществляется с помощью специальной программы, **компилятора**, являющимся разновидностью **транслятора**. Следует отметить, что язык программирования и компилятор языка программирования — это разные вещи. Язык программирования представляет собой математическое описание структур и возможностей, теоретически по этому описанию каждый может создать компилятор, поддерживающий данный язык. Поэтому для одного и того же языка, в частности языка Си, разработано большое количество компиляторов. К сожалению или к счастью, компиляторы не являются тождественными между собой: они добавляют в язык дополнительные возможности, по разному обрабатывают некоторые ошибочные ситуации и т.п. Хуже того, сам язык не является 100% стабильным: его **стандарт** модифицировался несколько раз и не все компиляторы поддерживают все стандартны. В связи с этим, с одной стороны есть заинтересованность промышленного программирования использовать все возможности современных стандартов и компиляторов, с другой — если нет уверенности в жесткой привязке к стандарту и компилятору, следует использовать самый базовый вариант языка, который будет строго обрабатываться любыми компиляторами.

В-третьих, любая программа есть инструмент обработки некоторых **данных**, процесс исполнения программы — обработка конкретных данных. Соответственно, для программы существуют **входные** и **выходные** данные, то есть те данные, которые необходимо обработать и те данные, которые получаются в результате обработки. В лекционном курсе также вводится понятие **алгоритм** и **черный ящик**. Теоретически и входные, и выходные данные могут отсутствовать. Следует отметить, что для программы входные и выходные данные могут как поступать с устройств ввода (клавиатура, мышь) и выводиться на устройства вывода (монитор, принтер) — то есть вводиться и получаться пользователем, так и передаваться в другие программы.

В-четвертых, ключевым инструментом программирования вообще, и программирования на языках высокого уровня в частности, особенно на языке Си, является **функция**. Ее также можно назвать подпрограммой (в некоторых языках разделяют два вида подпрограмм, процедуры и функции, но в Си такого деления не предусмотрено, поэтому данные слова можно использовать как синонимы). Функция представляет собой программную реализацию некоторой задачи, малую программу, обрабатывающую входные данные и выдающую выходные некоторым конкретизированным образом. Например, функция может вычислять квадратный корень, собственно вводить и выводить данные особым образом, изменять состояние программы и системы, и т.д. и т.п.

Входные и выходные данные функции могут быть как реально входными и выходными данными, вводимыми пользователем и выводимыми для нее, так и передаваемые в другие функции.

Источниками функций служат операционная система компьютера, **библиотеки функций** языка программирования высокого уровня, позволяющих за одну инструкцию языка программирования высокого уровня выполнить некоторое количество машинных команд (в частности, функции ввода и вывода данных не являются элементарными машинными командами, но их выполнение с помощью стандартных функций Си осуществляется обычно за одну инструкцию программы), так и сами программы. По сути программа состоит из функций, вызывающих друг друга все глубже и глубже, пока в конце концов не встретится вызов машинных инструкций.

Hello, World! на языке Си

Традиционно изучение языка программирования начинается с рассмотрения программы, выводящей сообщение *Hello, World!* ("Привет, Мир!"). На языке Си данная программа может выглядеть следующим образом.

```
1 #include <stdio.h>
2
3 int main ()
4 {
5     printf ("Hello,␣World!\n");
6     return 0;
7 }
```

Это далеко не единственный способ реализовать на языке Си вывод такого сообщения на экран, но именно данный вариант будет полезен для дальнейшего изучения. Не вдаваясь в фундаментальные детали, которые поясняются в теоретическом курсе, неформально и простым языком поясним каждую строчку данной программы.

Строка 1 содержит команду, позволяющую сделать возможным *вызов ряда функций стандартной библиотеки*, в том числе корректно использовать инструкцию строки 5.

`#include` — команда подключить нужный файл, файл `stdio.h` отвечает за функции ввода и вывода. Как правило такие инструкции должны следовать в начале программы.

Следует обратить внимание на то обстоятельство, что если опустить данную строку, то весьма вероятно, что программа будет работать и во многих случаях даже работать правильно, что создает ложное впечатление ненужности данной строки. Однако это далеко не всегда так, та *ошибочная программа, которая сработала правильно на одном компиляторе или при одном запуске, может не сработать на другом или при другом запуске.*

Строка 2 оставлена пустой в качестве разделителя: выше нее подключение библиотек, ниже — программа.

Строка 3 содержит создание *главной функции* программы. Это такая функция, которой передается управление при запуске программы. Суть строки в следующем: `int` — *тип данных* (о них речь пойдет ниже) выходного параметра функции, слово `main` — обязательное имя главной функции, пустые круглые скобки — отсутствие входных данных функции. Почему главная функция имеет возвращаемое значение, описывается в пояснении к строке 6.

Строки 4–7 содержат **тело** функции `main` — те команды, которые будут исполнены при вызове функции.

Строки 4 и 7 представляют собой **операторные скобки**. Вообще говоря, тело функции представляет собой набор инструкций, **операторов**, исполняемых последовательно, друг за другом. Операторные скобки в данном случае ограничивают тело функции, хотя с другой стороны они "превращают" набор операторов в один составной оператор, поэтому синтаксически тело функции есть один **составной оператор**. Операторы внутри тела функции разделяются точкой с запятой.

Строка 5 содержит вызов функции `printf`, выводящей на экран строку. Выводимая строка является **аргументом** функции, аргументы функции записываются в круглых скобках. Строка — строковые данные — записываются в двойных кавычках. Особый набор символов '`\n`' представляет собой команду перевода строки, чтобы следующий вывод данных, в том числе системных, осуществлялся с новой строки, а не в продолжении данной. В Си вызов функции является самостоятельным оператором, точка с запятой завершает его.

Строка 6 содержит оператор возврата значения из функции `main`. Как требует стандарт Си это должно быть целое число, что подчеркивается типом `int` возвращаемого значения в строке 3. Хотя на первый взгляд

для функции `main` имеют смысл только те выходные данные, что передаются пользователю, никакая функция программы `main` не вызывает, это не совсем так. Дело в том, что у всякой программы есть код возврата, передаваемый в операционную систему. Именно этот код возврата передается в данной строке. Ноль означает, что программа завершена без ошибок.

Следующим шагом следует уяснить, что правилом хорошего тона программирования является указание в программе комментариев. Комментарии — участки текста программы, которые не являются исполняемыми, а представляют собой текст для автора и читателя программы. Например, де-факто обязательным является указание того, что представляет собой программа (название, цель создания, какую задачу и как программа решает и т.п.), а также кто ее автор. Комментарии в Си записываются между комбинациями символов '`/*`' и '`*/`'. Комментарии могут быть частью строки, в одну или в несколько строк.

```
/* Sample C Hello, World! programm
 * for C programing workshop cource
 * (C) John J. Smith
 */

#include <stdio.h>

/* main types "Hello, World!" */

int main ()
{
    printf ("Hello,␣World!\n");
    return 0;
}
```

Отметим еще одно важное обстоятельство. Си является языком свободного переноса строк. Вообще говоря, только инструкцию `#include` критично записать в отдельной строке, поэтому такие программы как

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

или

```
1 #include <stdio.h>
2 int
3 main
4 ()
5 {printf
6 ("Hello,␣World!\n");
7 return 0;}
```

будут эквиваленты исходной, но трудночитаемыми. В связи с этим следует придерживаться ряда правил форматирования и оформления исходных текстов программ.

- Каждый оператор (инструкцию) следует записывать в отдельную строку.
- Следует использовать пробелы и отступы, например, внутри операторных скобок следует делать отступ в 4 пробела.
- Следует использовать столбцы. В данном случае операторные скобки указаны друг под другом, хотя такой вариант

```
1 #include <stdio.h>
2
3 int main () {
4     printf("Hello,␣World!\n");
5     return 0;
6 }
```

тоже допустим.

- Многострочный текстовый комментарий следует начинать с символа `*`, чтобы отличать его от "закомментированного", то есть спрятанного на время отладки в комментарий кода.

Создание программ на ЯВУ

Существуют разные способы создания программ на языках программирования высокого уровня. Все они включают несколько этапов: написание исходного кода (текста) программы, попытки компиляции и исправление синтаксических ошибок в тексте, обнаруженных компилятором, препятствующих компиляции или предупреждений, свидетельствующих о возможном наличии смысловой ошибки, тестирование и отладку, то есть нахождение и исправление смысловых ошибок, приводящих к неверному результату работы программы, документирование и публикация.

Написание и правка исходного кода программы может осуществляться с помощью любого текстового редактора (не путать с текстовым процессором, который использует нетекстовый формат!). Компиляция — с помощью вызова программы-компилятора из командной строки, пошаговая отладка — с помощью вызова специальной программы-отладчика.

Существуют также специальные программы, **интегрированные окружения рабочего стола** (IDE, Integrated Desktop Environment), представляющие собой комбинацию средств текстового редактора, вызова компилятора и визуализации процесса отладки. В отличие от языков программирования, такие программы, как компиляторы, отладчики и интегрированные окружения рабочего стола, являются быстроменяющимися продуктами, поэтому их детальное описание в настоящем пособии нецелесообразно. Укажем ключевые принципы, которыми следует руководствоваться при решении лабораторных работ.

1. В IDE следует создавать т.н. “проект” (может также использоваться название “решение” или иное). Недостаточно просто создать Си-файл! Если в начале программы будут простыми и состоять из одного файла, то в дальнейшем будут создаваться многофайловые программы и иначе как с помощью проектов IDE не сможет их обработать. Кроме того, функционал IDE при работе с одним Си-файлом может быть строго ограничен.
2. В проект действительно следует включать хотя бы один файл исходного кода программы (как правило, это происходит автоматически). Это файл с расширением `.c`.
3. Для большинства задач курса следует использовать тип проекта “консольное приложение” (Console application). В базовом курсе изучается создание программ, работающих в текстовом режиме, в текстовом терминале, с консольным вводом и выводом, в противоположность более сложным графическим приложениям.
4. Как правило, IDE для каждого проекта создают две конфигурации компилятора — отладочную (Debug) и публикационную (Release). Они отличаются тем, то в первом случае возможен пошаговый анализ программы, а во втором созданная программа будет работать быстрее благодаря подключению оптимизации в настройках компилятора. Каждую из этих конфигураций следует использовать по назначению.
5. **Полезно проследить, чтобы в настройках компилятора были подключены все предупреждения. Язык Си допускает большую свободу синтаксиса, которая может привести к труднообнаружимым смысловым ошибкам. Значительная часть таких ошибок может быть выявлена с помощью предупреждений.**
6. Крайне желательно также настроить компилятор на строгое следование стандарту языка Си, C89 или C90, в некоторых случаях можно использовать C99, но не все компиляторы его поддерживают.
7. Следует учесть, что IDE не будет перекомпилировать программу, если ее исходный код не был изменен. Однако, в некоторых случаях это необходимо. Например, если программа не содержала синтаксических ошибок, но компилятор выдал предупреждения, то исполняемый файл — скорее всего неверно работающий — будет получен все равно. Если предупреждения не исправить сразу, то они исчезнут из “поля зрения” программиста, а при следующей попытке компиляции они не будут выведены вновь — ведь исполняемый файл уже получен и программа не будет перекомпилироваться. Обычно в IDE присутствуют возможности сборки проекта, учитывающие наличие изменений, и пересборки, проводящую полную компиляцию в любом случае.

Примеры простейших программ

Рассмотрим следующую программу.

```
1 /* Вычисление суммы двух чисел
2  * (C) Авторы пособия, 2017
3  * Входные данные: два целых числа
4  * Выходные данные: сумма введенных чисел
5  */
6
```

```
7  #include <stdio.h>
8
9  int main ()
10 {
11     int n1, n2, sum;
12
13     printf ("Введите␣два␣целых␣числа:␣");
14     scanf ("%d%d", &n1, &n2);
15
16     sum = n1+n2;
17
18     printf ("Сумма␣введенных␣чисел␣равна␣%d\n", sum);
19
20     return 0;
21 }
```

Как видно из описания, программа запрашивает на вход два целых числа и выводит их сумму. Прежде всего следует обратить внимание на **строку 11**: в этой строке **объявляются переменные**. Конкретно в этой строке сказано, что будет три переменных — `n1`, `n2` и `sum`, в которых будут храниться целые числа: это определяется ключевым словом `int` перед именем переменной.

Переменную можно понимать как поименованную область (**ячейку**) памяти, способ обратиться к данным, находящимся в памяти, не по безликому и неизвестному в момент написания программы цифровому адресу, а по говорящему, понятному имени. Переменная в программировании имеет и схожее с переменной в математике значение: это некоторый символ (слово), которому сопоставляется значение (данные), причем это значение может быть изменено в процессе работы программы.

Все данные, с которыми работает программа, написанная на языке высокого уровня, различаются по **типам**, то есть формату представления. С точки зрения хранения все данные представляют собой двоичные числа, от типа зависит, во-первых, размер одного экземпляра данных, во-вторых, способ их обработки. В теоретическом курсе иллюстрируется, что даже сложение целых и вещественных чисел производится принципиально разным образом.

Строка 13 выводит сообщение на экран знакомым образом, **строка 14 вводит** значения переменных с клавиатуры. Здесь следует обратить внимание на три момента. Во-первых, ввод осуществляется с помощью функции `scanf`. Во-вторых, первым аргументом функции является особая строка, в которой указывается какие переменные и в каком формате вводятся. В данном случае вводятся 2 целочисленных переменных, поэтому для каждой из них указана комбинация `%d`. В-третьих, вторым и третьим аргументами являются вводимые переменные, но предваренные знаком `&`. Это связано с тем, что когда переменная используется без такого знака, то Си рассматривает ее **значение**, а в данном случае функции `scanf` требуется не значение переменной, а адрес той ячейки памяти, куда нужно записать данные. Чтобы получить этот адрес используется знак `&`.

Строка 16 ответственна за собственно вычисление суммы. Здесь указывается, что в переменную `sum` следует записать результат вычисления арифметического выражения: знак `=` означает **присваивание**, то есть задание нового значения переменной, справа от знака равенства указывается присваиваемое значение, в данном случае сумма (`+`) значений двух переменных.

Строка 18 выводит результат на экран с помощью функции `printf`. Здесь внутри строки также используется комбинация `%d`, которая будет заменена на значение целочисленной переменной, являющейся очередным аргументом функции `printf`. Эта функция использует именно значение переменной, поэтому указание знака `&` не требуется.

Строки 12, **15**, **17** и **19** оставлены пустыми, чтобы визуально разделить участки кода, ответственные за разные процессы: объявление переменных, ввод данных, вычисление результата, вывод данных, завершение работы программы.

Строго говоря, без переменной `sum` и строки 16 можно было обойтись, программа с таким же функционалом имела бы вид

```
1  /* Вычисление суммы двух чисел
2   * (C) Авторы пособия, 2017
3   * Входные данные: два целых числа
4   * Выходные данные: сумма введенных чисел
5   */
6
7  #include <stdio.h>
8
9  int main ()
```

```
10 {
11     int n1, n2;
12
13     printf ("Введите␣два␣целых␣числа:␣");
14     scanf ("%d%d", &n1, &n2);
15
16     printf ("Сумма␣введенных␣чисел␣равна␣%d\n", n1+n2);
17
18     return 0;
19 }
```

Такой подход укорачивает программу, снижает ее сложность как за счет количества строк, так и за счет количества переменных — хотя, скорее всего, не экономит память, так как для хранения результата сложения все равно потребуется ячейка, и не улучшает быстродействие. С другой стороны, здесь вывод результата (часть, относящаяся к пользовательскому интерфейсу программы) и вычисления (рабочая часть) объединены, что, вообще говоря, является плохой практикой программирования, затрудняющей модификацию и сопровождение программ, о чем еще будет вестись речь в этом курсе.

Рассмотрим аналогичную программу, но работающую уже с вещественными числами.

```
1  /* Вычисление суммы двух чисел
2   * (C) Авторы пособия, 2017
3   * Входные данные: два вещественных числа
4   * Выходные данные: сумма введенных чисел
5   */
6
7  #include <stdio.h>
8
9  int main ()
10 {
11     double x1, x2, sum;
12
13     printf ("Введите␣два␣вещественных␣числа:␣");
14     scanf ("%lf%lf", &x1, &x2);
15
16     sum = x1+x2;
17
18     printf ("Сумма␣введенных␣чисел␣равна␣%lf\n", sum);
19
20     return 0;
21 }
```

Программа претерпела ряд принципиальных изменений и одно изменение, связанное с удобством восприятия. В **строке 11** вместо `int` указано `double` — это объявление вещественной переменной. В **строках 14** и **18** вместо `%d` указывается `%lf`, что соответствует вещественному числу. Последнее связано с тем, что хотя Си предоставляет свободу выбора переменных программисту, имена должны быть удобными для восприятия. Кроме того, существуют некоторые традиции, согласно которым (как и в математике), переменные n, m и т.п. — целые, а x, y и т.п. — вещественные. В остальном код программы не отличается.

Рассмотрим другие арифметические действия.

```
1  /* Иллюстрация основных арифметических действий
2   * (C) Авторы пособия, 2017
3   * Входные данные: два целых числа
4   * Выходные данные: сумма, разность, произведение, отношение,
5   *                  неполное частное, остаток
6   */
7
8  #include <stdio.h>
9
10 int main ()
11 {
12     int n1, n2, sum, sub, mul, div, mod;
13     double frac;
14
15     printf ("Введите␣два␣целых␣числа:␣");
```

```
16      scanf ("%d%d", &n1, &n2);
17
18      sum = n1+n2;
19      sub = n1-n2;
20      mul = n1*n2;
21      div = n1/n2;
22      mod = n1%n2;
23
24      frac = (double) n1/n2;
25
26      printf ("Сумма␣введенных␣чисел␣равна␣%d\n"
27              "Разность␣введенных␣чисел␣равна␣%d\n"
28              "Произведение␣введенных␣чисел␣равно␣%d\n"
29              "Отношение␣введенных␣чисел␣равно␣%lf\n"
30              "Неполное␣частое␣введенных␣чисел␣равно␣%d\n"
31              "Остаток␣от␣деления␣чисел␣равен␣%d\n",
32              sum, sub, mul, frac, div, mod
33             );
34
35      return 0;
36  }
```

В **строках 18–22** проводятся арифметические действия над целыми числами: сложение (+), вычитание (-), умножение (*), нахождение неполного частного (/), нахождение остатка от деления (%). Для вещественных чисел арифметические операции выполняются аналогично, только вычисление остатка неприменимо. Однако получить вещественное число в результате прямого деления целых чисел невозможно. Вместо этого в **строке 24** сначала первое число делается вещественным, то есть *приводится к соответствующему типу.* Если хотя бы один из операндов арифметического действия — вещественное число, то и результат будет вещественным числом. В принципе, того же самого можно было достичь с помощью `frac = n1 / (double) n2;` или наивным образом `frac = 1.0 * n1 / n2;`.

На **строки 26–33** разорван вызов функции вывода результата на экран. На самом деле у этой функции по-прежнему первый параметр-строка, просто она разделена на несколько для удобства: следующие подряд строки Си объединяет в одну, остальные параметры — список выводимых значений. При этом в строках сделаны отступы для соблюдения правила столбцов, по этой же причине круглые скобки расположены одна под другой.
.

Лабораторная работа №1
Знакомство с IDE языка Си.

Задание 1.1. *Добейтесь работоспособности выбранного IDE: запустите его на публичном компьютере, установите на домашнем компьютере (при наличии), установите компилятор и отладчик.*

Задание 1.2. *Создайте проект, разберитесь с настройками компилятора. Изучите процесс создания проекта.* При выполнении данной задачи следует иметь в виду, что каждый проект — исходный код программы. Следует использовать говорящие имена проектов, соответствующие названию программы, при этом название программы должно говорить само за себя, то есть из его названия должно прямо следовать, что программа делает.

Задание 1.3. *Реализуйте программу "Hello, World!", добейтесь, чтобы она компилировалась и исполнялась в обоих конфигурациях. Изучите процесс компиляции, перекомпиляции и запуска программы средствами IDE. Изучите реакцию IDE на нулевой и ненулевой код возврата функции* `main`.

Задание 1.4. *Модифицируйте программу таким образом, чтобы сообщение "Привет, Мир!" выводилось на русском языке кириллическими символами.* При работе с некоторыми ОС, например, Windows, может возникнуть проблема: редактор IDE, скорее всего, будет работать в одной кодировке символов (Windows-1251), а консольный терминал — в другой (DOS-866), поэтому кириллические символы отобразятся некорректно. Необходимо либо настроить IDE на кодировку DOS-866, либо переключить терминал на кодировку Windows-1251 в начале запуска программы. Делается это с помощью добавления двух строк кода:

```
1  #include <stdio.h>
2  #include <locale.h>
```

```
3
4  int main()
5  {
6      setlocale(LC_CTYPE, "Russian");
7      printf("Привет,␣Мир!\n");
8      return 0;
9  }
```

2-я строка подключает необходимую библиотеку функций, 6-я — вызов функции по переключению настройки локали. Такой способ не совсем корректен, так как использует системно-зависимые конструкции.

Задание 1.5. *Запустите пример иллюстрации всех арифметических вычислений. Добейтесь его компиляции и исполнения в обоих конфигурациях. Запустите программу через отладчик. Изучите процесс пошагового исполнения программы. Изучите процесс наблюдения за изменением переменных в процессе отладки программы.*

§1.2. Некоторые тонкости

Типы данных

Основные типы данных в Си делятся на две группы: целочисленные и с плавающей точкой. Среди целочисленных выделяется символьный тип `char`, представляющий собой код одного символа строго в 1 байт. В Си коды символов и сами символы с точки зрения типов данных неразличимы, различие определяется на стадии обработки.

Целочисленные типы различаются по длине и наличию или отсутствию знака. Базовый знаковый целочисленный тип обозначается `int`, беззнаковый — `unsigned int` или просто `unsigned`. Сколько именно байтов отводится под хранение данных этого типа в стандарте языка не конкретизируется, это зависит от архитектуры компьютера и компилятора. Типы другой длины рассмотрены в теоретическом курсе.

Базовый тип для вычислений с плавающей точкой — `double`.

Отдельного строкового типа нет, по сути строка представляет собой **массив** (набор) символов, условно данный тип можно обозначать как `char *` или `char []` или в некоторых случаях это `const char *`, подробное объяснение смысла данного обозначения приводится в теоретическом курсе.

Получение данных в программе

Один из способов задания данных в программу — указание их непосредственно в коде программы. Второй способ — получение данных с помощью операций ввода с внешнего устройства. В примерах данными, указанными непосредственно в коде программы, являются строки. Также можно указывать и числовые данные. Например, программа вычисления суммы чисел могла бы выглядеть так

```
1  /* Вычисление суммы двух чисел
2   * (C) Авторы пособия, 2017
3   * Входные данные: два целых числа
4   * Выходные данные: сумма введенных чисел
5   */
6
7  #include <stdio.h>
8
9  int main ()
10 {
11     int n1 = 1, n2 = 2, sum;
12
13     sum = n1+n2;
14
15     printf ("Сумма␣чисел␣равна␣%d\n", sum);
16
17     return 0;
18 }
```

или даже так

```
1  /* Вычисление суммы двух чисел
2   * (C) Авторы пособия, 2017
3   * Входные данные: два целых числа
```

```
 * Выходные данные: сумма введенных чисел
 */

#include <stdio.h>

int main ()
{
    printf ("Сумма␣чисел␣равна␣%d\n", 1+2);

    return 0;
}
```

В данном случае это плохой подход: ясно, что редкая программа будет всегда применяться к одним и тем же данным, необходимо иметь возможность получать результат для разных входных данных, а если эти данные будут жестко прописаны в программу, то при смене данных программу придется изменять, что, как правило, неудобно для конечного пользователя. Однако, часто требуется явное задание данных в код программы.

Тип данных, вводимых непосредственно в код программы, определяется самими данными. Такие данные также называются `константами`. Так, если данные содержат только цифры — то это целочисленные данные, при этом числа, начинающиеся с `0`, считаются записанными в восьмеричной системе счисления, с комбинации `0x` — в шестнадцатеричной, с любой другой цифры — в десятичной. Например `12` — число 12, `-1` — число -1, `010` — число 8, `0xA` — число 10.

Если число содержит десятичную точку или порядок — комбинацию вида `e` / `E` и целое число (возможно со знаком) — то это считается числом с плавающей точкой. Запись вида m`E`p означает число $m \cdot 10^p$. Например, `1.0` — число 1, но не целое, `-1.5` — число -1.5, `1.2e-3` — число `0.0012`, `1e0` — число 1, но тоже не целое. В теоретическом курсе вводятся суффиксы для числовых констант, позволяющие определять длину типа.

Символьные данные записываются в апострофах, например `'A'`, `'1'` — символ 1, но не целое число 1, код символа 1 может отличаться от числа 1.

Строковые данные записываются в кавычках, при этом строку данных можно разорвать на несколько заключенных в кавычки, компилятор их объединит. Это необходимо, чтобы разбить строку данных на несколько строк программы. Например, следующие инструкции эквивалентны

```
printf ("Hello,␣World!\n");

printf ("Hello,␣"   "World!\n");

printf ("Hel" "lo,␣World!\n");

printf ("Hello,␣"
        "World!\n"
       );
```

Второй и третий пример являются примерами плохого кода, четвертый — хорошего, просто для столь коротких строк в таком разрыве нет необходимости. За исключением собственно содержимого строк и символов, где следует закрыть кавычку и апостроф до завершения строки программы, Си переносы строки приравнивает к пробелам, а множественные пробелы и переносы строки считает за один (это, конечно, не относится к пробелам внутри строковых констант).

Переменные

В процессе работы программы все данные хранятся в памяти. Данные, жестко записанные в код программы, хранятся вместе с кодом программы, для данных, вводимых с устройств ввода, и для хранения результатов вычислений, необходимо резервировать ячейки памяти. Делается это с помощью создания **переменных** — особых синтаксических объектов языка Си. Переменная характеризуется именем и типом данных, который хранится в соответствующей ячейке памяти.

Имя переменной выбирает программист исходя из того, что именно по смыслу хранится в данной переменой. В качестве имен переменных могут выступать последовательности букв латинского алфавита и цифр, но начинаться имя переменной должно с буквы. При этом буквы разного регистра считаются разными, поэтому `var`, `Var`, `VAR` и `vAr` — разные переменные. Компилятор автоматически определяет номер (адрес) ячейки памяти, где будут храниться данные задаваемой переменной. Чтобы это произошло необходимо переменную **объявить**. Делается это с помощью специальной инструкции, представляющей собой указание типа данных и перечисление имен объявляемых переменных через запятую. Завершатся объявление переменных одного типа точкой с запятой. Пример:

```
double x, y, z;

int i1, i2;

char c;
```

По стандарту Си объявление переменных может производится только в начале операторного блока, то есть сразу после открывающейся фигурной скобки, до списка операторов. Хотя объявления переменных похожи на операторы, строго говоря они таковыми не являются.

Перед использованием переменных их необходимо **инициализировать**, то есть задать значение. Неинициализированная переменная является источником **непредсказуемого** поведения, т.к. вообще говоря нигде не гарантируется, что ее значение будет нулевое, во многих случаях это не так и на практике. *Очень важно следить, чтобы в программе не было использования значений неинициализированных переменных. Современные компиляторы имеют возможность подключить предупреждения о неинициализированных переменных, на него следует обязательно обращать внимание.*

Задание значения переменных осуществляется несколькими способами.

Первый — задать значения при объявлении переменных с помощью инициализатора. Он указывается после имени переменной в виде знака равно и задаваемого значения.

```
double x = 1.0, y = 0.0, z = 0.0;

int i1 = 256, i2 = 128;
```

Вообще говоря, в инициализаторе можно использовать и *выражения* (см. ниже), вычисляя значения переменных через уже инициализированные

```
double x = 1.0, y = 2.0, z = x + y;
```

Второй способ состоит в том, чтобы **присвоить** значение переменной во время работы программы. Так можно не только задать значение переменной, но и изменить его. Конструкция аналогична инициализатору, используется знак равенства:

```
double x = 1.0, y = 2.0, z, r;

/* ... */

r = 5.0;
z = x + y;
```

Третий способ — ввод данных с устройств ввода, о нем пойдет речь ниже.

Следует отметить, что инициализировать выражения переменных можно не обязательно данными строго объявленного типа: целое число автоматически и однозначно приводится к числу с плавающей точкой, поэтому

```
double x = 1;
```

суть корректная конструкция.

При работе с переменными следует руководствоваться следующими соглашениями:

- Имена переменных должны быть говорящими сами за себя, то есть соответствовать смыслу хранимых данных. Использовать однобуквенные переменные за редким исключением не рекомендуется.
- Для каждой переменной должно быть четко сформулировано (в комментарии или хотя бы в голове), что именно хранится в данной переменной.
- Если нет веских причин сделать обратное, в Си принято использовать имена переменных, записанных строчными буквами.

Заметим, что неиспользуемая переменная не является смысловой ошибкой сама по себе, но компиляторы могут быть настроены на предупреждение об их наличии в программе: вероятнее всего, такую переменную следует убрать как излишнюю, но есть и вариант, что программист объявил ее с какой-то целью и “забыл” реализовать соответствующий код.

Ввод и вывод.

Ввод и вывод осуществляется с помощью функций `printf` и `scanf`. Это функции *форматированного ввода и вывода*, так как они осуществляют вывод и вывод данных в определенном формате.

Следует отметить, что почти все функции в Си имеют фиксированное количество аргументов, каждый из которых должен быть определенного типа. Для `printf` и `scanf` это не так: только первый аргумент фиксирован, он должен быть и должен быть строкой, это **строка формата**. Остальные аргументы опциональны, их количество не детерминировано, они могут иметь любой тип. В контексте данной функции эти аргументы называются *полями*. В качестве полей могут выступать любые данные — константы, переменные, выражения. То, как именно будут прочитаны и выведены поля, определяется содержимым строки формата. Для каждого поля в строке следует отнести комбинацию символа `%` и указания на то какого типа данных очередное поле и в каком формате будет выведено указанное поле.

Для вывода данных типа `int` поле `%d`, типа `unsigned int` — `u` для вывода в десятичной системе, `o`, `x` и `X` — для вывода в восьмеричной, шестнадцатеричной и шестнадцатеричной заглавными буквами соответственно.

Для вывода вещественных чисел можно использовать `%lf` — вывод в простом формате, `%lE` или `%le` — для вывода в экспоненциальной форме (выведется заглавная или строчная E соответственно) или `%lG` или `%lg` для выбора более оптимального варианта между `%lf` и `lE`/`le`.

Для вывода знака % следует указать `%%`.

Кроме задания собственно выводимого типа данных, после `%` можно регулировать формат выводимых данных.

Пример.

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int m;
6      double b, c;
7
8      m = 7;
9
10     printf("m␣=␣%d\n",m);
11     printf("m␣=␣%3d\n",m);
12     printf("m␣=␣%03d\n",m);
13     printf("m␣=␣%+03d\n",m);
14
15     b = 1.23456789;
16     c = 1.23456789E5;
17
18     printf("b␣=␣%3.4lf,␣b␣=␣%3.4le,␣b␣=␣%3.3lE,␣b␣=␣%3.5lG␣c␣=␣%3.5lG␣\n", b, b, b, b, c);
19
20     return 0;
21 }
```

Выведет

```
m = 7
m =   7
m = 007
m = +07
b = 1.2346, b = 1.2346e+00, b = 1.235E+00, b = 1.2346 c = 1.2346E+05
```

В **строке 11** указано вывести не мене трех цифр, недостающие дополняются пробелами, в **строке 12** — нулями, а в **строке 13** дополнительно выводится знак + даже для положительных чисел. В **строке 18** для вещественных чисел комбинация из двух чисел через точку имеет следующий смысл: первое число по-прежнему число выводимых цифр, дополняемых пробелами или нулями, второе — число цифр, выводимых после точки (для `lf` и `le`/`lE`) или общее число значащих цифр (для `lg`/`lG`).

Форматы ввода для `scanf` аналогичны, хотя `X`, `lE`, `lG` не используются, а `lf`, `le`, `lg` имеют одинаковый смысл. Однако, у функции есть существенное различие при передаче полей, куда будет осуществляться запись введенных данных. Во-первых, это должны быть именно переменные — то есть ячейки памяти, куда можно записать значение. Во-вторых, в функцию нужно передавать не сами переменные, а адреса ячеек памяти, соответствующих переменным. Дело в том, что в Си параметры передаются по значению, то есть значения переменных или иных данных, указанных в качестве аргумента функции, копируется из области памяти, где хранятся данные вызывающей функции, в другую область памяти, где хранятся данные вызываемой функции. Для того, чтобы получить из переменной ее адрес, необходимо предварить символом `&`.

Следует обратить внимание, что вообще говоря стандарт Си полностью оставляет на совести программиста правильность указания длины и спецификатора, равно как и количества полей. Только современный компи-

лятор, настроенный на вывод соответствующих предупреждений, сообщит что что-то сделано не так. Вообще говоря, даже в этом случае он обязан создать работающую программу. Однако, в случае `printf` результат вывода может оказаться неожиданным:

```
#include <stdio.h>

int main()
{
    int i = 1, j = -3;
    double a = 3.7, b = 10;

    printf ("%lf\n", i);
    printf ("%d␣%d␣%d\n", a);

    return 0;

}
```

может вывести что-то похожее на

```
0.000000
4195827 -338725056 2147483639
```

В первом случае так распозналась целочисленная переменная `i` как типа `double`, во втором — переменная `a` типа `double` "распалась" на две целочисленные (обычно `double` занимает вдвое больше байт, чем `int`), а в качестве третьего значения вообще вывелся т.н. "мусор", то есть какие-то значение, случайно оказавшиеся в памяти в соответствующем поиску поля функцией `printf` месте.

Вообще говоря, результат работы такой программы `непредсказуем`.

Для `scanf` ситуация еще хуже: если `printf` просто выведет "странные" значения, то эта функция может *перезаписать* данные, которые не должны быть перезаписаны. Это могут быть как другие переменные, так и часть кода программы, что приведет к неверным результатам или краху ее работы в странный момент.

```
#include <stdio.h>

int main()
{
    int i = 1, j = -3;

    scanf ("%lf", &i);

    printf ("%d␣%d\n", i, j);

    return 0;
}
```

при попытке ввода 1.5 выведет что-то подобное

```
1.5
0 1073217536
```

не только переменная `i` не ввелась корректно, еще и нарушилась целостность переменной `j`.

Совсем неприятно, если забыть символ `&`: тогда значение переменной будет воспринято как адрес и в эту ячейку памяти будут записаны данные. Где будет эта ячейка памяти — непредсказуемо, это может быть одна из переменных программы, а может быть часть ее кода. В небезопасных операционных системах так может быть нарушена и целостность операционной системы. В самом неблагоприятном случае может оказаться, что введенные данные приведут к появлению в части кода программы верной, но нежелательной инструкции: например, уничтожения всех данных на компьютере. Вероятность того, что это произойдет случайно ничтожно мала, но такая ошибка может быть использована со злым умыслом.

Аналогичные неприятности могут возникнуть, если в функциях форматированного ввода и вывода указать больше полей в строке формата, чем передать значений или переменных для вывода и ввода: откуда и куда будут выводиться недостающие данные непредсказуемо.

При работе с функциями форматированного ввода и вывода следует соблюдать особую осторожность. *Важно получать и исправлять все предупреждения компилятора, касающиеся строки формата. Это важно для printf и критично для scanf.*

Ввод и вывод строк является более сложной задачей, ей посвящен соответствующий раздел.

Арифметические операции и выражения

Арифметические операции в Си — `+` (сложение), `-` (вычитание), `*` (умножение), `/` (неполное частное или отношение), `%` (получение остатка от деления). Операции могут быть применены как к данным (константам) и переменным, так и к результатам других операций, что позволяет записывать составные арифметические выражения, причем Си поддерживает общепринятый порядок (приоритет) выполнения операций — умножение и деление (в том числе получение остатка) выполняется раньше сложения и вычитания. Регулировать приоритет можно с помощью круглых скобок. Например

```
r1 = (x1 + x2) * x3;
r2 = x1 + x2 * x3;
r3 = x1 + (x2 * x3);

t1 = 1 / 3 + 5;
t2 = 1.0 / 3 + 5;
```

Выражения, присваиваемые в `r2` и `r3` одинаковы, в `r1` — отличается. В `t1` запишется 5, в `t2` — приближенное представление $5\frac{1}{3}$.

Данные, к которым применяются операции, называются **операндами**.

Операция во многих смыслах похожа на вызов функции — входные данные (операнды) обрабатываются с помощью некоторой подпрограммы, в результате чего получаются выходные данные (результат операции). С помощью **операций** образуются **выражения**, представляющие собой *способ преобразования данных*: фактически выражение тоже есть некоторые данные, определенного типа. Вычисление выражения можно представлять себе как последовательный процесс замены операций с операндами на результат применения операций к операндам, начиная с самой **приоритетной** операции. Тип данных выражения может отличаться от типов данных операндов, например, результат деления данных типа `int` на данные типа `double` имеет тип `double`.

Следует также отметить, что *присваивание также является операцией*. Ее приоритет ниже почти всех других операций, поэтому она выполняется последней. Отличие присваивания от арифметических операций в том, что она не только вычисляет некоторое значение, но еще и *изменяет значение в ячейке памяти*, то есть состояние системы и программы. Такое явление называется **побочным эффектом операции**. Пример.

```
1 int n = 5, m = 2;
2 double d = 1.5, r;
3 /* ... */
4
5 r = m + ((n + 2) * 3) / d + 2.5;
```

В **строке 4** вычисление осуществляется, например, следующим образом:

```
1 r = m + ((n + 2) * 3) / d + 2.5;
2 r = m + (7 * 3) / d + 2.5;
3 r = m + 21 / d + 2.5;
4 r = m + 19.5 + 2.5;
5 r = 21.5 + 2.5;
6 r = 24.0;
7 /* здесь следует применение побочных эффектов */
```

Вначале вычисления идут согласно скобкам, в **строке 4** при отсутствие скобок вычисления идут слева направо, последним шагом происходит изменение значения переменной `r`. Присваиваемое выражение имеет тип `double`. Следует учесть, что все результаты промежуточных вычислений хранятся в некоторых ячейках памяти, их можно рассматривать как *неявные переменные*.

На самом деле присваивание не только изменяет значение переменной, но возвращает присвоенное значение. Причем если при отсутствии скобок арифметические операции выполняются слева направо, то операции присваивания — справа налево, поэтому такое присваивание корректно:

```
a = b = c = 1;
```

До сих пор рассматривались операции, имеющие *два операнда*. Такие операции называются **бинарными**. Существуют и операции, имеющие *один операнд* — **унарные**. Например, существуют унарный минус `-`, результат применения которого к числу (данным), записанному справа от него, есть изменение знака числа, унарный плюс `+`, не меняющий число, но часто удобный для симметризации записи. Унарной операцией является и взятие адреса переменной — `&`, это операция с единственным операндом, переменной, адрес которой необходимо получить. Унарной операцией является и **приведение типа**, рассмотренное на примере получения вещественного отношения целых чисел: `(double) n1 / n2` — указание типа данных в скобках есть приведение типа.

Помимо традиционных арифметических операций в Си введены две особые операции **инкремента** `++` и **декремента** `--` — увеличения или уменьшения значения целочисленной переменной на единицу. Они могут применяться только к переменным и созданы для того, чтобы менять значения переменой *быстро* — как в плане скорости написания программы, так и в плане ее работы. Это унарные операции, но они присутствуют в двух вариантах — когда операнд записывается справа от операции (**префиксный инкремент** и **префиксный декремент**) и когда операнд записывается слева от знака операции (**постфиксный инкремент** и **постфиксный декремент**). В плане *побочных эффектов* они выполняют одну и ту же работу — изменяют значение переменной. Поэтому можно записать

```
int i = 1;

++i; /* теперь i = 2 */
i++; /* теперь i = 3 */

--i; /* теперь i = 2 */
i--; /* теперь i = 1 */
```

Различие между ними в том, какое значение *возвращает* данная операция: постфиксные возвращают старое значение переменной, префиксные — измененное. Поэтому

```
int i = 1, j;

j = ++i; /* теперь i = 2, j = 2 */
j = i++; /* теперь i = 3, j = 2 */
```

При этом префиксный инкремент и декремент — более быстрые в плане выполнения, они не требуют хранения предыдущего значения переменной, поэтому **всегда следует отдавать предпочтение префиксному варианту инкремента и декремента, если нет причины использовать постфиксный**, то есть если предыдущее значение переменной не надо вернуть в выражение.

Для удобства программирования в Си введены операции **арифметического присваивания** — `+=`, `-=`, `*=`, `/=`, `%=`. Рассмотрим их на примере `+=`, остальные ведут себя аналогично. Левым операндом такой операции должна быть переменная, правым некоторые данные. Операция увеличивает значение переменной на значение правого операнда, то есть `a+=b` то же самое, что `a = a+b`, но выполняется быстрее, так как происходит непосредственно в ячейке памяти `a`, без хранения промежуточного результата в неявной переменной и переноса его в нужную ячейку. Например,

```
int n = 1, m = 2;

n += 2;      /* Теперь n=3 */
m += n + 3;  /* Теперь m=8 */
```

Последнее действие выполняется именно таким образом, так как операции арифметического присваивания низкоприоритетны, то есть сначала будет применено сложение.

Следует отдавать предпочтение операциям арифметического присваивания, а не конструкциям вида `a=a+/*...*/`.

Операции арифметического присваивания возвращают присвоенное значение и выполняются справа налево, поэтому такая запись возможна.

```
int n = 2, m = 3;

n += m += 3;    /* сначала m = 6, затем n = 8 */
```

К сожалению, операции возвращают не переменную, поэтому запись вида

```
int n = 2, m = 3;

(n += m) += 3;    /* хотим сначала n = 5, затем n = 8, m неизменно */
```

является ошибочной в стандартном Си.

Неопределенное и непредсказуемое поведение

Синтаксис операций в Си представляется очень гибким и свободным, это позволяет писать громоздкие вычислительные конструкции. Отчасти для того, чтобы программисты не слишком увлекались этим, отчасти

потому, что Си — еще и язык, направленный на создание быстро работающих программ, некоторые вещи, касающиеся порядка и способа вычисления значения выражения не стандартизированы, создатели компиляторов имеют возможность самостоятельно выбрать более оптимальный путь.

Рассмотрим простой пример.

```
int i = 5, j;
j = ++i + ++i;
```

Чему равно `j`? Это зависит от того, как считать. Первый вариант, применяется левый инкремент, `i` становится равным 6, это значение возвращается, применяется правый инкремент, `i` становится 7, это значение возвращается `6+7` есть 13. Однако, реально скорее всего будет применен другой способ. Применяется один из инкрементов, это префиксный инкремент, это быстрейшая операция, она просто изменяет значение переменной `i`, т.е. в ячейке оказывается 6. Применяется второй инкремент, тоже меняет значение `i` — оно стало 7. Теперь складывается `i+i` — ответ 14.

Реально результат зависит от компилятора и его настроек.

Более того, вообще говоря Си не регламентирует порядок вычисления операндов:

```
int i = 5, j;
j = i++ + i;
```

Может быть сначала будет взято значение i (5, правый операнд), затем изменено значение `i` и взято предыдущее значение (5, левый операнд) — ответ 10? А может быть сначала будет взято значение `i` (5, левый операнд), а затем изменено и взято новое (6, правый операнд) — ответ 11?

Такое явление называется **неопределенным поведением**, имеется в виду то, что поведение программы не определено стандартом, результат определяется компилятором. Рассмотренное ранее **непредсказуемое** поведение связанное с использованием неинициализированных переменных, не определяется даже компилятором и зависит от состояния компьютера в момент запуска программы.

Чтобы избежать неопределенного поведения необходимо следовать таким правилам.

- Если эксперимент показал, что исследуемый код работает именно так, это еще не значит, что он будет работать именно так всегда: на другой платформе, компиляторе, при других настройках он может дать другой результат. Только строгое следование требованиям стандарта, описанию возможностей языка, математическому описанию его синтаксиса и т.п. гарантирует то, что программа даст предсказуемый результат.
- Необходимо подключать соответствующие предупреждения компилятора и исправлять их.
- Следует исключить более чем однократное изменения значения одной и той же переменной в одном выражении (с помощью инкремента, арифметического присваивания, присваивания и т.п.).
- Следует избегать использование значения переменой в том же выражении, где ее значение изменяется. Исключение — использование значения изменяемой переменной в правой части операции присваивания, здесь стандарт гарантирует, что изменение значения переменной произойдет после всех остальных вычислений.

Размер данных и переменных

В стандарте Си не декларируется **длина** для большинства основных типов данных, то есть количество байтов, отводимых для хранение переменной каждого типа. Гарантируется лишь, что тип `char` является строго однобайтовым. (Правда для `char` не гарантируется, является ли он знаковым или нет, для конкретизации нужно использовать `signed char` или `unsigned char`, впрочем, чаще этот тип используется не для вычислений, а для хранения кодов символов, знак которых не важен.)

Среди целочисленных типов выделяют “классический” (`int`), “короткий” (`short`), длинный (`long`), в некоторых компиляторах поддерживается и тип `long long`. В строке формата при форматированном вводе и выводе для них следует использовать формат поля `i`, `hi`, `hl`, `hL` и т.п. При этом вообще говоря часть или все эти типы могут совпадать по длине.

Базовым типом является `int`, так как его размер соответствует т.н. *машинному слову*, размеру единовременно обрабатываемых данных, размеру ячейки памяти, доступ к которой осуществляется за один акт, значения типа `int` выравниваются по ячейкам памяти, доступ к которым осуществляется быстрее всего и т.п. Если нет явных причин использовать другие целочисленные типы, в вычислениях следует использовать `int`, хотя в целях экономии при хранении больших количеств небольших целочисленных значений использовать `short`, если нужно, наоборот, представит большие значения — `long` и т.п.

Среди вещественных чисел различают стандартизированные типы одинарной точности (`float`, формат `f`), двойной точности (`double`, формат `lf`), и нестандартизированный формат повышенной точности (`long double`, формат `Lf`). В качестве базовых рекомендуется использовать вычисления с двойной точностью.

В лекционном курсе рассматриваются также суффиксы при задании констант различных типов данных.

Для того, чтобы определить реальный размер типа данных в процессе исполнения программы можно использовать унарную префиксную операцию `sizeof`. Она может быть применена как к имени типа, так и к данным. Например,

```
char c;
int i;

printf("%lu␣%lu␣%lu␣%lu", sizeof c, sizeof i, sizeof double, sizeof 'A');
```

Результат операции `sizeof` имеет тип, который обозначается как `size_t`. Вообще говоря, такого типа в самом языке Си нет, он приравнен к одному из беззнаковых целочисленных типов (самому большому), поэтому вероятнее всего строка для него будет `lu` или `Lu`, что, однако, нестандартизировано.

Математические функции

Как уже было отмечено, стандартной библиотекой языка Си предоставляется ряд функций. Для того, чтобы функции стали доступны в программе, необходимо подключить соответствующий заголовочный файл. При вызове функции указывается имя функции, затем в круглых скобках перечисляются ее **аргументы** через запятую.

Функция обычно также имеет **возвращаемое значение** — данные некоторого типа. В качестве аргументов функции могут выступать любые выражения соответствующего типа. Следует отметить, что и в качестве операнда может выступать выражение некоторого типа. Правда, если, как уже было отмечено, для одной и той же операции от типа операндов может зависеть как именно будет выполняться операция и какой будет тип результата операции, то в Си почти все функции имеют фиксированное количество и тип аргументов (исключение составляют функции неопределенного числа аргументов, такие как `printf` и `scanf`), а также тип возвращаемого значения (без исключений).

Таким образом, функции можно применять к результатам операции и наоборот, уровень вложенности выражений неограничен. Такой подход обеспечивает "естественность" выражений в Си — их аналогичность математическим выражениям, где используются числа, переменные, операции и функции.

Как и операции, функции с аргументами образуют выражения, представляют собой способ получения данных, при вычислении выражений функция с аргументами "заменяется" на результат применения некоторого алгоритма к аргументам.

Например, в файле `stdlib.h` (то есть для его использования в начале программы нужно указать

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

определена функция `abs`, функция нахождения модуля (абсолютной величины). Функция имеет единственный аргумент типа `int` и возвращает число типа `int`.

В файле `math.h` определен ряд математических функций. *Замечание: в некоторых системах, например GNU/linux, для их использования недостаточно просто записать* `#include <math.h>`*, может понадобиться подключить библиотеку* `m` *в настройках компилятора.*

Функция	Количество и тип аргументов	тип и смысл возвращаемого значения
`fabs`	1, `double`	`double`, абсолютная величина числа
`sin`	1, `double`	`double`, синус угла, заданного в радианах
`cos`	1, `double`	`double`, косинус угла, заданного в радианах
`tan`	1, `double`	`double`, тангенс угла, заданного в радианах
`exp`	1, `double`	`double`, экспонента
`log`	1, `double`	`double`, натуральный логарифм числа
`pow`	2, оба `double` (a, b)	`double`, возведение в степень a^b
`sqrt`	1, `double`	`double`, квадратный корень
`floor`	1, `double`	`double`, наибольшее целое число, не превосходящее аргумент
`round`	1, `double`	`double`, округление до ближайшего целого

Это далеко не полный перечень функций. Вообще говоря, в рамках курса не стоит задача обучить *всем функциям всех библиотек*. В теоретическом курсе даются необходимые знания для того, чтобы самостоятельно изучать функции библиотек по документации.

Обратите внимание на разницу между функциями `abs` и `fabs`: одна работает с целыми числами, другая с вещественными. Отметим, что в Си существует **неявное** приведение типа, оно используется, если операнд

операции или аргумент функции имеет тип, отличный от требуемого, но может быть приведен однозначно. Например, вызвать `fabs` для целого числа с получением вещественного результата можно без дополнительных ухищрений — например, `fabs(-3)` Си приведет -3 к -3.0, а функция вернет 3.0. Вызвать `abs` для вещественного числа в зависимости от компилятора или нельзя, или можно с неявным приведением — потерей дробной части, что может привести с неожиданным результатам: `abs(-1.5)` есть целое число 1.

Также следует обратить внимание, что такие функции как `floor` и `round` возвращают вещественное по формату, хотя и целое по содержанию число. Если нужно иметь именно целочисленное по типу данных значение, можно воспользоваться явным приведением:

```
int i;
double d;

/* ... */

i = (int) round (d);
```

Завершая разговор о выражениях, отметим, что в Си выражение является **оператором**, то есть минимальной исполняемой инструкцией программы. Однако, имеет смысл исполнение только тех выражений, которые имеют побочные эффекты, то есть либо меняют значение переменных, либо выводят какие-то данные и т.п. При этом, использовать возвращаемое значение выражения не обязательно. Конечно, вызов

```
sin(x);
```

без использования значения функции `sin` лишен смысла, а вот вызов

```
printf ("Hello,␣World!\n");
```

вполне оправдан даже если возвращаемое значение функции `printf` — целое число, количество выведенных байтов — не используется. Аналогично,

```
x + 2;
```

есть верный оператор-выражение, который вообще говоря ничего не делает. Хотя с точки зрения Си такой оператор корректен, если это опечатка и имелось в виду

```
x += 2;
```

программа окажется со смысловой ошибкой. *Современные компиляторы могут предупреждать о выражениях без побочных эффектов. Следует рассматривать и исправлять эти предупреждения.*

Лабораторная работа №2

Знакомство с языком Си.

Задание 2.1. *Напишите следующие программы (данные вводятся с клавиатуры)*

(a) вычисление модуля (абсолютной величины) целого числа;

(b) вычисление модуля (абсолютной величины) вещественного числа;

(c) вычисление синуса и косинуса введенного угла в радианах;

(d) вычисление синуса и косинуса введенного угла в градусах.

С помощью строки формата добейтесь того, чтобы выводимый результат результат выглядел красиво и читабельно. Указание. Для получения числа *pi* можно использовать т.н. *символическую константу* M_PI, заданную в файле `math.h` как псевдоним для соответствующей явной константы.

Задание 2.2. *Напишите программу, позволяющую определить размер переменных различных типов и результатов операций с помощью операции sizeof.*

Задание 2.3. *Выполните ряд ошибочных примеров из текста.* Обратите внимание на предупреждения, выдаваемые компилятором.

(a) Неверное указание типа данных и количества полей при вводе и выводе.

(b) Неопределенное поведение.

(c) Использование неициализированных переменных.

(d) Наличие неиспользуемых переменных.

(e) Использование библиотечной функции без подключения соответствующего файла.

Задание 2.4. *Даны следующие переменные*

```
double a = 2, b = 3, c = -1;
int i = 5, j = 2;
```

Определите тип и значение выражений.

(a) `a + b`

(b) `a + b - 1.0`

(c) `sin (a * M_PI) / j`

(d) `pow(i, j)`

(e) `i / j + c`

Глава 2. Основы структурного программирования

§2.1. Основные принципы создания и оформления программ

Общие замечания

При создании программ следует руководствоваться рядом принципов, рекомендаций и соглашений.

- Прежде чем написать программу, следует четко определить, что конкретно она делает, какие данные она получает на вход, что выдает на выходе. Если программист не может что-либо сформулировать на повседневном языке, то программист не сможет это реализовать на языке программирования.

  После получения словесной постановки задачи — сути черного ящика, выявления входных и выходных данных, — следует определить как именно программа преобразует входные данные в выходные математическим образом. *Рекомендуется начинать написание программы с соответствующего комментария.*

  Следующий шаг — определение компьютерного формата представления входных и выходных данных (количество и тип соответствующих переменных), а также алгоритмической (пошаговой) реализации математического преобразования, выявление промежуточных данных (количество и тип вспомогательных переменных)

  Далее следует собственно написание кода программы. При этом следует разделить интерфейсную часть (отвечающую за взаимодействие программы с пользователем) и рабочую часть (то есть собственно реализуемый алгоритм).

- Всякая программа должна быть качественно оформлена. Программа пишется не только для компьютера, но и для человека, который будет ее читать — проверять, сопровождать, модифицировать. Даже если этим человеком будет лишь автор программы, через некоторое время у человека есть высокий шанс забыть тонкости и детали создания кода и столкнуться с необходимостью восприятия своей программы как чужой. Поэтому программа должна быть написана хорошо читаемой. Это также очень важно и для процесса отладки, поиска ошибок. Код программы следует читать и редактировать, как и в случае с сочинением, первичный код не более, чем черновик. При прочтении могут быть как выявлены незаметные на первый взгляд ошибки, так и улучшена производительность и читаемость кода.

  Программа должна быть снабжена заголовочным комментарием, поясняющим кто является автором программы и что эта программа делает, а также пояснением ключевых нетривиальных мест кода. Комментарий должен быть предложением, а не обрывком фразы.

  В коде программы следует грамотно переносить строки (одна инструкция — одна строка, или делать больше разрывов строк для наглядности). Строка должна целиком помещаться на экран (обычно это не более 60–70 символов, даже если современные мониторы позволяют отобразить более длинные строки, превышать это значение не следует, так как возможности восприятия человеком все равно ограничены).

  Следует отделять части кода, отвечающие за различные этапы работы программы, пустыми строками, а также следить за ровными столбцами везде, где это возможно. В частности, вложенные в фигурные скобки блоки кода (например, тело функции `main`) следует делать с отступом в 4 пробела (это "**правило четырех пробелов**"), во многих случаях столбцы (отступы) следует соблюдать при разрывах длинных строк кода, при кодировании однотипных выражений разной длины (выравнивание, например, по знаку операции присваивания, комментариям и т.п.). Сравните

```
int k; /* Что делает k. */
unsigned long number; /* Опишите, что делает number. */
int l; /* Что делает l. */
k = 5; /* Здесь идет комментарий. */
numset(str); /* Здесь второй комментарий. */
l = k; /* А здесь третий. */
```

читается хуже, чем такой:

```
int k;                /* Что делает k. */
unsigned long number; /* Опишите, что делает number. */
int l;                /* Что делает l. */

k = 5;       /* Здесь идет комментарий. */
numset(str); /* Здесь второй комментарий. */
l = k;       /* А здесь третий. */
```

- Все имена, вводимые программистом (переменных, программы) должны быть говорящими. Более того, следует руководствоваться **принципом наименьшей неожиданности**: если программа названа "синус", она не должна считать квадратный корень. При использовании имен переменных следует учитывать некоторые соглашения по типам данных.
  - переменные, чьи имена начинаются с `i`, `j`, `k`, `l`, `n`, `m` — целые;
  - переменные, чьи имена начинаются с `a`, `b`, `c`, `d`, `e`, `f`, `x`, `y`, `p`, `q`, `r` — вещественные;
  - переменные, чьи имена начинаются с `c` — символьные;
  - переменные, чьи имена начинаются с `s` — строковые.
  - переменные, чьи имена начинаются с `b` — логические.

  Как видно, правила не жесткие (важнее использовать говорящие имена). Также следует учесть, что переменная `i` используется как счетчик, `s` может использоваться для целочисленного размера и т.п., но будет очень странно, если в программе встретится целочисленный `x` или вещественный `k`.
- Программа должна сообщать пользователю необходимую информацию, а именно: что требуется ввести, что именно выведено, если произошла ошибка — в чем она состоит как пользователь может ее исправить и т.п..
- Следует экономить машинные ресурсы: время и память. Если проблема экономии памяти на первых этапах обучения вряд ли актуальна (например, отказ от объявления переменной в пользу более сложного выражения, как уже было сказано, память не экономит), то задача экономии времени может встать довольно быстро: необходимо избегать повторяющихся вычислений, проверок условий и т.п.

Линейные программы

Следующая лабораторная работа посвящена линейным программам, т.е. программам, не содержащим условий и циклов, последовательность исполнения инструкций которой не зависит от входных данных. Во всех задачах следует исключить использование условного оператора.

Лабораторная работа №3
Линейные программы

Задание 3.1. *Дано трехзначное целое число. В нем*

(a) зачеркнули первую слева цифру и приписали ее справа;

(b) зачеркнули первую цифру справа и приписали ее слева.

Вывести полученное число. Для получения цифр числа использовать вычисление остатка, а не логарифмирование и потенцирование.

Задание 3.2. *С начала суток прошло n секунд (n — целое). Найти количество минут и секунд, прошедших с начала последнего часа. Вывести в формате мм:сс.* ($n = 11107$ — 3 часа, 5 минут, 7 секунд; вывод: 05:07).

Задание 3.3. *Даны координаты трех вершин треугольника:* $(x1, y1)$, $(x2, y2)$, $(x3, y3)$. *Найти его*

(a) периметр

(b) площадь.

Задание 3.4. *Даны вещественные положительные числа a, b, c. На прямоугольнике размера $a \times b$ размещено максимально возможное количество квадратов со стороной c (без наложений). Найти*

(a) количество квадратов, размещенных на прямоугольнике;

(b) площадь незанятой части прямоугольника;

(c) долю занятой части (процент заполненности квадратами) прямоугольника.

Основы представления числовых данных

В теоретическом курсе описан формат представления числовых данных: беззнаковое целое, целое со знаком в дополнительном коде, числа с плавающей точкой в формате $m \cdot 2^p$, где m и p — целые числа со знаком, мантисса и порядок, в двоичной системе счисления.

Лабораторная работа №4

Представление чисел в компьютере

Задание 4.1. *Предположим, что имеется некоторый целочисленный двухбайтовый тип. Каков результат следующих операций в данном типе:*

(a) $32000 \cdot 2$;

(b) $300 \cdot (-200)$;

(c) $65535 + 2$;

Рассчитать без компьютера отдельно для знакового и беззнакового типа. Учесть, что отрицательное число в беззнаковом типе превратится в некоторое положительное, слишком большое положительное в знаковом — в отрицательное.

Задание 4.2. *Даны следующие числа (записаны в десятичной системе счисления)*

(a) 128

(b) 78

(c) 3245

Найти сумму их десятичных и шестнадцатеричных цифр. Обратить внимание, что сумма цифр представления любой системы счисления может быть записана в любой системе счисления: записать сумму десятичных цифр в шестнадцатеричной системе и наоборот.

Задание 4.3. *Рассмотрим гипотетический двухбайтовый (не-IEEE) формат чисел с плавающей точкой. Пусть 6 бит отведено под порядок, 10 — под мантиссу (целые числа со знаком в дополнительном коде). Записать, как будут кодироваться следующие числа (каждый из 16 бит).*

(a) 1;

(b) 32;

(c) -32;

(d) 100;

(e) 0.2;

(f) π.

Задание 4.4. *Определить результат вычисления в типе данных из предыдущей задачи*

(a) 1000+1;

(b) -12345+5.

Задание 4.5. *Придумать и отобразить в Си программе примеры переполнения (знакового и беззнакового) для целых чисел, потери точности и переполнения порядка для вещественных чисел с реальными типами данных языка Си.*

§2.2. Условия, ветвления и циклы

Условные оператор и операции сравнения

Проверка условий является важным элементом программирования. Уже отмечалось, что функция представляет собой совокупность операторов, разделенных точкой с запятой. **Процесс исполнения функции есть процесс последовательного исполнения этих операторов.** Во многих случаев в зависимости о входных данных (точнее часто и в зависимости от результатов их предварительной обработки) нужно исполнить тот или иной код, то есть разные операторы или группы операторов.

Эту задачу решает условный оператор. У него две формы: полная

```
if ( /* выражение */ )
    /* оператор_1 */;
else
    /* оператор_2 */;
```

и неполная.

```
if ( /* выражение */ ) /* оператор */;
```

Суть оператора в следующем: выражение, записанное в круглых скобках, рассматривается как условие.

Если оно истинно, то полная форма оператора выполняет `оператор_1`, `оператор_2` игнорируется, если ложно — игнорируется `оператор_1`, выполняется `оператор_2`. Далее управление передается следующему за условным оператору.

Краткая форма выполняет единственный оператор-аргумент тогда и только тогда, когда условие истинно, по окончании работы в любом случае передает управление следующему за условным оператору.

Разумеется, оператор-аргумент может быть составным: *в Си там где можно оказать один оператор, можно указать и операторный блок.*

Условные выражения можно формировать, например, с помощью **операций сравнения чисел**: строго меньше (`<`), строго больше (`>`), меньше либо равно (`<=`), больше либо равно (`>=`), не равно (`!=`), равно (`==`). Если первые 4 операции достаточно естественны, а для понимания пятой лишь следует иметь в виду, что `!` в Си есть символ отрицания, то на равенство нужно обратить особое внимание: *не следует путать равенство и присваивание.*

```
1  if (a < b) c = a + b;
2
3  if (x == y)
4      printf("Числа␣равны\n");
5  else
6      printf("Числа␣не␣равны\n");
7
8  if (n >= m)
9  {
10     x = pow (n, m);
11     y = pow (m, n);
12 }
13 else
14 {
15     x = pow (m, n);
16     y = pow (n, m);
17 }
```

Обратите внимание на следование *правилу четырех пробелов* для операторов, “вложенных” в `if`, а также на наличие точки с запятой в строке 4 и отсутствию ее в строке 12.

При использовании условного оператора следует учитывать ряд обстоятельств.

- Условные операторы могут быть вложенными. Обычно это работает нормально, но в некоторых случаях может вызывать неоднозначное прочтение: операторы следует не только корректно форматировать, но и указывать фигурные скобки, даже если оператор один. Компилятор может предупреждать о неоднозначности прочтения.

```
/* В этом примере else относится ко второму if
 * отступ вводит в заблуждение человека,
 * но игнорируется компилятором
 */
```

```
if (cond1)
    if (cond2)
        statement_2;
else
    statement_3;

/* Это тот же самый код,
 * только по другому корректно() форматирован
 */
if (cond1)
    if (cond2)
        statement_2;
    else
        statement_3;

/* А здесь else корректно отнесен
 * к первому оператору if
 */
if (cond1)
{
    if (cond2)
        statement_2;
}
else
    statement_3;
```

- Часто вложенные условные операторы можно оформлять в виде цепочки `if...else if...else`.

```
/* Сложный для восприятия вариант */
if (a == b)
    printf ("a␣=␣b\n");
else
    if (a == c)
        printf ("a␣=␣c\n");
    else
        if (a == d)
            printf ("a␣=␣d\n");
        else
            printf ("Совпадений␣не␣найдено\n");

/* Тот же самый код, но форматированный по другому */
if (a == b)
    printf ("a␣=␣b\n");
else if (a == c)
    printf ("a␣=␣c\n");
else if (a == d)
    printf ("a␣=␣d\n");
else
    printf ("Совпадений␣не␣найдено\n");
```

- Внутри операторного блока — после открывающейся фигурной скобки — можно объявлять переменные. Только эти переменные будут видны лишь внутри данного операторного блока, после использовать их нельзя.
- Важная особенность Си: **в качестве условия может выступать выражение любого типа**. При этом истиной считается любое ненулевое значение, ложью только ноль. Поэтому **нельзя путать** присваивание и равенство. Запись типа `if (n = 2)` не только изменит значение переменной `n`, но и создаст тождественно верное условие — ведь присваивание возвращает присвоенное значение, а 2 есть истина. Современные компиляторы могут выдавать предупреждение о том, что присваивание используется как сравнение.

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

```
int main()
{
    int n;

    printf ("Введите␣целое␣число\n");
    scanf ("%d", &n);

    if (n = 2)                          /* тождественно истинное условие        */
        printf ("Число␣равно␣2\n");     /* для всех n будет выведена эта строка */
    else
        printf ("Число␣не␣равно␣2\n")   /* недостижимый код                     */

    return 0;
}
```

- В Си нет отдельного логического типа. Все операции сравнения возвращают значение типа `int`, либо 0 (ложь), либо 1 (истина). Тип `int` выбран потому, что он наиболее оптимален в смысле скорости доступа и выравнивания по блокам памяти.
- Последнее обстоятельство делает бессмысленным сравнение на не совпадение с нулем. "`if`(a != 0)"то же самое, что `if (a)`: **условный оператор как раз и проверяет, не является ли выражение нулем**.

  Первая запись создаст дополнительное действие — вычислить значение выражения `a != 0` (получится 0 или 1), а затем проверить, равно ли 0 полученное выражение, вместо того, чтобы просто проверить, равно ли само `a` нулю.

  Конечно, во многих случаях запись сравнения с нулем выглядит более понятной и наглядной, например, когда она соответствует сути алгоритма или формулы, то есть если надо действительно проверить на (не)равенство нулю некоторой математической величины. В таких случаях указание `!=0` оправдано.

  Однако в других моментах, например, когда проверяется, что переменная `a` задана (имеет нетривиальное значение, определено, существует, верно) или когда проверяется результат, возвращаемый *логической функцией*, то есть функцией, которая сама по себе проверяет некоторое условие, то запись, подобная `if(cond)` ("если верно условие"), `if(is_ok(expr))` ("если выражение проходит проверку"), `if(p)` ("если `p` задано") и т.п. являются более соответствующие сути вещей.
- Сравнивать вещественные и целые числа между собой можно, а вот **сравнение знакового и беззнакового целого — неопределенное поведение.** Действительно, если сравнивать -1 (знаковый) и 5 (беззнаковый) — кто из них больше? Как действовать — привести -1 к беззнаковому и получить самое большое число данного типа (например, в однобайтовом типе -1 знаковый есть 11111111, при прочтении как беззнакового получится 255, что больше 5) или 5 привести к знаковому (оно останется равным 5, что больше -1)?

  В данном случае корректный результат дал второй способ, но если сравнивать -1 и слишком больше, чтобы быть приведенным к знаковому, беззнаковое число, то правильным будет первый способ. Компилятор не может предугадать, какой способ сработает для заранее неизвестных ему данных, но может предупредить о неопределенности сравнения знакового и беззнакового целого.

Логические операции

Часто приходится проверять не одно условие, а одновременное выполнение нескольких условий или выполнение одного из условий и т.п. Для этого используются **логические операции**, которые на вход берут операнды любых типов, в частности `int`, как результат операций сравнения, на выходе возвращают значение типа `int`, либо 0 (ложь), либо 1 (истина).

Операции следующие: логическое И (`&&`, истинно только когда оба операнда истинны), логическое ИЛИ (`||`, истинно когда хотя бы один из операндов — истина), логическое НЕ (`!`, отрицание, меняет истину на ложь и наоборот).

Не следует путать логические операции и побитовые: использование `&` вместо `&&` и `|` вместо `||` часто некорректно, хотя во многих случаях визуально может работать правильно.

Это корректный код проверки совпадения значения трех переменных

```
if (a == b && b == c)
    printf ("Числа␣равны\n");
else
    printf ("Числа␣не␣равны\n");
```

Этот код **не корректен**

```
if (a == b == c)  /* Ошибка !!!! */
    printf ("Числа␣равны\n");
else
    printf ("Числа␣не␣равны\n");
```

Это верно записанная программа на Си, но она не проверяет равенство трех чисел: вместо этого проверяется равенство результата сравнения `a` и `b` — 0 или 1 — и `c`, то есть истиной будет комбинация $a = 2$, $b = 2$, $c = 1$ или $a = 2$, $b = 3$, $c = 0$.

Логические операции имеют более низкий приоритет, чем операции сравнения, поэтому дополнительные скобки в строке `if (a == b && b == c)` не требуются. В то же время `&&` имеет более высокий приоритет, чем `||`, хотя в любом случае при комбинировании нескольких различных бинарных логических операций в одно условие рекомендуется использовать скобки во избежание ошибок.

Благодаря операции отрицания `if (a == 0)` можно записывать как `if (!a)`.

Тернарное условие

Часто проверить условие можно и без использования условного оператора. Для этого в Си есть тернарная **условная операция** `?:`. У нее три операнда: условие и два выражения, первое есть значение операции при истинности условия, второе — при ложности. Например, найти какое число больше можно таким образом:

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

но более коротким будет код

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

а пример из предыдущего параграфа можно сократить до

```
    printf (a == b && b == c ? "Числа␣равны\n" : "Числа␣не␣равны\n");
```

Условная операция сокращает код и рекомендуется к использованию.

Оператор выбора

Хотя использование цепочки `if...else if...else` удобно, часто удобнее воспользоваться **оператором выбора**.

```
if (i == 1)
{
     /* случай i == 1 */
}
else if (i == 2)
{
     /* случай i == 2 */
}
else if (i == 3)
{
     /* случай i == 3 */
}
else if (i == 4)
{
    /* случай i == 4 */
}
else
{
    /* остальные случаи */
}
```

записывается как

```
switch ( /* выражение */ )
{
   /* объявления */

   case /* константа 1 */ :

      /* операторы, выполняемые если выражение == константа 1 */
      break;

   case /* константа 2 */ :

      /* операторы, выполняемые если выражение == константа 2 */
      break;

   /* ... */

   default :

      /* операторы, выполняемые если выражение не равно ни одной константе */
}
```

использование break чаще всего необходимо, так как иначе выполнение программы не прервется только потому, что встретился следующий помеченный оператор и будет исполняться до самого конца оператора-аргумента switch (или до первого break). Это используется, например, в случаях, когда двум выбираемым значениям соответствует одно действие (случай ИЛИ для множественных `if`):

```
if (i == 1)
{
     /* случай i == 1 */
}
else if (i == 2 || i == 3)
{
     /* случай i == 2 или 3 */
}
else if (i == 4 || i == 5 || i == 6)
{
     /* случай i == 4 или 5 или 6 */
}
else
{
    /* остальные случаи */
}
```

превращается в

```
switch (i)
{
    case 1:
            /* случай i == 1 */
            break;
    case 2:
    case 3:
            /* случай i == 2 или 3 */
            break;
    case 4:
    case 5:
    case 6:
            /* случай i == 4 или 5 или 6 */
            break;
    default:
            /* остальные случаи */
}
```

Оператор `switch` может генерировать более быстрый код, так как вместо проверки условий сразу передает управления нужному блоку, однако имеет свои ограничения: выражение должно быть целочисленным, а после `case` должна следовать константа (явные целочисленные данные).

Проверка корректности ввода

Использование `scanf` связано с определенным риском: пользователь может ввести данные, которые не могут быть преобразованы в соответствующий формат (ввести буквы вместо числа и т.п.). В связи с этим важно проверять, все ли числа были прочитаны корректно. *Функция* `scanf` *возвращает число фактически корректно прочитанных полей.* В случае неверного ввода можно, например, завершить программу с ошибкой. Например, для ввода двух чисел

```
1  /* Нахождение максимума двух чисел
2   * (C) Авторы пособия, 2017
3   * Входные данные: два целых числа
4   * Выходные данные: наибольшее из двух чисел
5   */
6
7  #include <stdio.h>
8
9  int main ()
10 {
11     int n1, n2, max;
12
13     printf ("Введите␣два␣целых␣числа:␣");
14     if (scanf ("%d%d", &n1, &n2) != 2)
15     {
16         printf ("Неверный␣ввод,␣следует␣ввести␣два␣целых␣числа\n");
17         return 1;
18     }
19
20     max = n1 > n2 ? n1 : n2;
21
22     printf ("Наибольшее␣из␣введенных␣чисел␣%d\n", max);
23
24     return 0;
25 }
```

В строке 17 оператор return не только возвращает значение, но и завершает работу функции main, а вместе с ней и всей программы, с кодом возврата 1, что сообщает системе о том, что программа завершилась с ошибкой.

Лабораторная работа №5

Ветвления и условия

Задание 5.1. *Дан номер года (положительное целое число). Определить количество дней в этом году.* Обычный год содержит 365 дней, високосный — 366 дней. Високосным считается год, делящийся на 4, за исключением тех годов, которые делятся на 100 и не делятся на 400 (например, годы 300, 1300 и 1900 не являются високосными, а 1200 и 2000 — являются).

Задание 5.2. *Известны год, номер месяца и день рождения каждого из двух человек. Определить, кто из них старше.*

Задание 5.3. *Вводится число от 1 до 99 — сумма в рублях. Вывести эту сумму строкой, с указанием количества рублей и правильным согласованием падежей.* (Например, 20 — “двадцать рублей”, 32 — “тридцать два рубля”, 41 — “сорок один рубль”.) Указание. Использовать `switch`.

Задание 5.4. *Вводится число от 1 до 99 — возраст в годах. Вывести возраст строкой, с указанием количества лет, правильным согласованием падежей и слова год.* (Например, 20 — “двадцать лет”, 32 — “тридцать два года”, 41 — “сорок один год”.) Указание. Использовать `switch`.

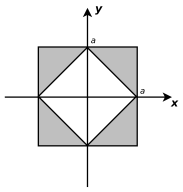
Задание 5.5. *Даны длины сторон двух треугольников. Определить подобны ли эти треугольники.* Указание: стороны могут вводиться в произвольном порядке, предполагается сортировка трех чисел с помощью `if`.

Задание 5.6. *Даны длины трех отрезков a, b, c. Определить вид треугольника, образованного отрезками такой длины: тупоугольный, прямоугольный, остроугольный, вырожденный в отрезок, несуществующий.*

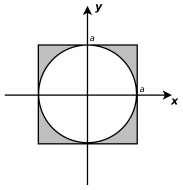
Задание 5.7. *Даны четыре вещественных числа — координаты противоположных углов прямоугольника* $(x1, y1)$, $(x2, y2)$, *стороны которого параллельны осям координат, и координаты пробной точки* (x, y). *Определить, находится ли точка внутри прямоугольника.* Учесть, что углы могут вводиться в любом порядке, неизвестно какой из них выше, какой левее.

Задание 5.8. *Даны области, определенные параметром a и точка с координатами* (x, y). *Определить, находится ли точка внутри затемненной обрасти.*

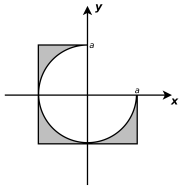
(a)



(b)



(c)



Задание 5.9. *Написать программу анализа квадратного уравнения* $ax^2 + bx + c = 0$. *Рассмотреть все возможные варианты: два вещественных корня, один кратный корень, два комплексных корня, линейное уравнение с одним корнем* ($a = 0$, $b \neq 0$), *уравнение, не имеющее корней* ($a = 0$, $b = 0$, $c \neq 0$), *любое вещественное число является корнем* ($a = b = c = 0$). *Вывести полученный случай и корни (если есть).* Указание. Исключить повторения вычислений (дискриминанта, корня из него и т.п.) и проверок одного и того же условия.

Замечание. В задачах 5–8 рассмотреть вариант, учитывающий, что ответ на поставленный вопрос может быть дан лишь с некоторой точностью. Использовать вводимую пользователем относительную погрешность.

Упрощение логических выражений

Лабораторная работа №6

Логическое выражения

Задание 6.1. *Даны следующие переменные*

```
double a = 2, b = 3, c = -1;
int i = 5, j = 2;
```

Определите тип и значение следующих выражений. Указание: отрицание имеет более высокий приоритет, чем арифметические операции и операции сравнения.

(a) `a + b < c`

(b) `a == i - 1 && j`

(c) `!(c == -1)`

(d) `!j`

(e) `a && b || c`

(f) `a == 1 || b == 3 && c == 5`

(g) `a && !c == 0`

Задание 6.2. *Используя логические законы, упростите выражения.* Указание. Отрицание строго неравенства есть обратно нестрогое неравенство и наоборот (по крайней мере это верно для целых чисел).

(a) `!(m < n && !(n > 1))`

(b) `!(m == n || !(n > 1) && n != 0)`

Особенности сравнения чисел с плавающей точкой

При вычислениях с плавающей точкой неизбежны **ошибки округления**, то есть потеря точности. Это делает проверку на равенство и неравенство вещественных чисел некорректной операцией: математически равные числа могут отличаться, например, на последнюю цифру мантиссы, что сделает их неравными с точки зрения компьютера.

Можно использовать сравнение с некоторой точностью ε: считать два числа a и b равными, если $|a - b| < \varepsilon$, однако числа a и b в разных случаях могут быть разных порядков и использовать один и тот же ε для всех случаев некорректно. Например, при сравнении чисел первого десятка отличие в 0.1 может быть существенным, а для чисел $\sim 10^{100}$ разница в 10^{90} может оказаться на грани точности.

Поэтому, вместо упомянутой выше **абсолютной погрешности** лучше (хотя и не идеально, но идеального решения просто не существует) использовать **относительную погрешность**:

$$\left|\frac{a-b}{a}\right| < \varepsilon$$

или даже

$$\left|\frac{a-b}{\max(a,b)}\right| < \varepsilon$$

если порядок этих чисел тоже может отличаться существенно.

Кроме того, при работе с типами с плавающей точкой в Си следует учитывать возможность, что результат операции окажется бесконечностью (INF) или не-числом (NaN). Кстати, не-число не равно ни какому числу, в том числе самому себе, результат любой операции сравнения с ним будет ложью. Поэтому, для вещественных чисел `!(a<b)` не то же самое, что `a>=b`. Кроме того, код

```
if (a < b)
    printf ("a␣меньше␣b\n");
else if (a > b)
    printf ("a␣больше␣b\n");
else if (a == b)
    printf ("a␣равно␣b\n");
```

не перебирает все возможные ситуации, нужен еще случай

```
if (a < b)
    printf ("a␣меньше␣b\n");
else if (a > b)
    printf ("a␣больше␣b\n");
else if (a == b)
    printf ("a␣равно␣b\n");
else
    printf ("a␣или␣b␣-␣не␣число\n);
```

Бывает полезно сгенерировать не число или бесконечность. В некоторых стандартах Си существуют соответствующие константы (`NAN` и `INF`), в общем случае не-число получается как `0.0/0.0`, бесконечность — как `1.0/0.0`, однако проверка на то, что значение является не-числом осуществляется не с помощью `a==NAN` (это всегда ложь), а с помощью `a!=a` (это истинно если `a` — не-число).

Циклы

В Си существует три оператора цикла — с предусловием, с постусловием и оператор for. Цикл с предусловием записывается как

```
while ( /* выражение */ ) /* оператор */;
```

с постусловием

```
do { /* список операторов */ } while ( /* выражение */ );
```

Разумеется, цикл с предусловием также может применяться к составному оператору

```
while ( /* выражение */ ) { /* список операторов */ }
```

В качестве условия может выступать выражение любого типа данных. Тело цикла исполняется до тех пор, пока условие истинно (то есть не ноль).

Цикл с предусловием сначала проверяет условие, затем, если оно истинно, исполняет оператор. Цикл с постусловием сначала исполняет оператор, затем проверяет условие. Таким образом, цикл с постусловием приведет к выполнению тела цикла хотя бы один раз. Для цикла с постусловием обязательны операторные скобки.

Оператор, выполняемый циклически, называется **телом цикла**.

Третий оператор цикла аналогичен по названию и своему предназначению циклам со счетчиком других языков программирования, однако имеет более широкий синтаксис:

```
for ( /* Инициализатор */ ; /* Условие */ ; /* Итератор */ ) /* Оператор */;
```

по сути является циклом с предусловием и эквивалентен следующей конструкции:

```
/* Инициализатор */ ;
while ( /* Условие */ )
{
    /* Итератор */;
    /* Оператор */;
}
```

В качестве инициализатора, условия и итератора могут выступать любые выражения. Цикл сначала вычисляет инициализатор, затем выполняет цикл с предусловием, вычисляя итератор после каждого прохода. От инициализатора и итератора ожидается наличие побочных эффектов, на самом деле любое из трех выражений может быть оставлено пустым, тогда итератор и инициализатор просто не вычисляются, а пустое условие считается всегда истинным. Обычное использование цикла `for` — со счетчиком:

```
for (i = 0; i < 10; i++) printf ("%d␣%d\n", i, i*i);
```

В Си для целочисленных счетчиков типично использование нестрого неравенства. Разумеется, вместо `i++` можно указать, например, `i--` или `i+=2`. Разумеется, счетчик может быть и вещественным, а может изменяться не только в итераторе, но и в теле цикла.

При работе с циклами следует учесть ряд обстоятельств.

- С помощью оператора цикла можно создать программу, работа которой никогда не завершится. Например, указать 1 в качестве условия или составить условие таким образом, что сколько бы раз тело цикла не исполнялось, оно не станет ложным: **в теле цикла (или в хотя бы итераторе в случае `for`) следует обеспечить изменение условия**. Простейшие случаи бесконечного цикла:

  ```
  while (1);

  do {} while (1);

  for (;;);
  ```

- Цикл с постусловием всегда выполняет тело цикла хотя бы один раз. В то же время цикл с предусловием может быть не выполнен ни разу. В программировании чаще встречаются ситуации, когда при некоторых данных не нужно выполнение тела цикла, чем гарантированное требование по крайней однократного выполнения тела. Поэтому рекомендуется использовать цикл с предусловием, за исключением ситуаций, когда использование цикла с постусловием четко обосновано.
- С условиями в циклах (как и в условных операторах) связан риск перепутать равенство и присваивание. Следующий цикл бесконечен.

  ```
  while (i = 1) do_something();
  ```

- Переменная, определенная в теле цикла, не видна в условии цикла, следующий код также создает бесконечный цикл.

```
int i = 1;
while (i)
{
    int i = 0;
}
```

- Операторы цикла и условные операторы могут быть вложенными друг в друга. **Структурное программирование** основано на том, что любой алгоритм можно закодировать с помощью трех **структур управления** — последовательности, ветвления и цикла. Последовательность в Си определяется последовательностью записи операторов через точку с запятой.

  Одна из основных идей структурного программирования в том, что структуры управления регулирует порядок выполнения операторов или операторных блоков, то есть составных операторов Си. Программа состоит из вложенных блоков кода, выполняющих некоторое обособленное действие, порядок исполнения и повторения исполнения этих блоков определяется условиями

- При оформлении операторов цикла также следует использовать правило четырех пробелов — делать соответствующий отступ для тела цикла, чтобы визуализировать уровни вложенности и сделать легко видимым какой оператор тела к какому циклу или условию относится.

- Оператор `continue` приводит к непосредственному переходу к следующей итерации тела цикла, при этом для цикла `for` происходит вычисление итератора, оператор `break` прерывает исполнение цикла (самого внутреннего) и передает управление следующему за данным оператором цикла оператору. Использование данных операторов нарушает принципы структурного программирования и не рекомендуется за исключением случаев, когда это оправданно производительностью или читабельностью кода. В учебных задачах их лучше избегать.

Лабораторная работа №7

Циклы

Задание 7.1. *Дано целое положительное число. Найти*

(a) сумму его цифр

(b) наибольшую из его цифр

(c) наименьшую из его цифр

Замечание. Использовать цикл `while` для последовательного получения и отсечения цифр, возведение в степень исключить. Решить в двух вариантах: для цифр только десятичного представления, для цифр представления в заданной системе счисления. В последнем случае учесть, что ввод числа и его p-ичных цифр (суммы цифр) может и должен осуществляться в десятичной системе счисления.

Задание 7.2. *Дано целое положительное число. Визуализировать проверку его делимости на 3 с помощью признака делимости: циклическое сложение всех цифр числа с получением нового числа, к которому применяем тот же признак, до тех пор пока не получится однозначное число, для которого делимость проверяется условием совпадения с делящейся на 3 цифрой. На каждом шаге программа должна отображать складываемые цифры числа и их сумму, на последнем шаге вывести ответ.* Цель — не проверить делимость на 3 (для этого нужно использовать операцию вычисления остатка), а именно визуализировать признак.

Задание 7.3. *С клавиатуры вводится последовательность строго положительных вещественных чисел. Заранее количество вводимых чисел неизвестно, признак конца ввода — 0 (в последовательность не входит). Найти*

(a) наибольшее из этих чисел;

(b) наименьшее из этих чисел;

(c) наибольшее (наименьшее) из четных чисел списка;

(d) наибольшее (наименьшее) среди чисел списка, кратных заданному числу (вводится предварительно);

(e) наиболее близкое значение число списка (в смысле модуля разности значений) к заданному числу (вводится предварительно);

(f) сумму этих чисел;

(g) среднее арифметическое этих чисел;

(h) среднее геометрическое этих чисел;

(i) наибольшее и второе по величине число;

(j) наименьшее и второе снизу число;

(k) длину самого длинного участка возрастания чисел;

(l) длину самого длинного участка убывания чисел;

(m) длину самого длинного участка монотонного следования чисел, с указанием того, был это участок возрастания или убывания.

Более одного вводимого числа списка одновременно в памяти не хранить. Там где это возможно по условию, решить в двух вариантах: когда ответом является значение (число), и когда ответом является номер введенного числа.

Задание 7.4. *Составить программу для возведения заданного натурального числа в третью степень используя следующую закономерность*

$$\begin{aligned} 0^3 &= 0 \\ 1^3 &= 1 \\ 2^3 &= 3+5 \\ 3^3 &= 7+9+11 \\ 4^3 &= 13+15+17+19 \\ 5^3 &= 21+23+25+27+29 \end{aligned}$$

Задание 7.5. *Напечатать таблицу значений функции на отрезке $[a,b]$ с шагом h.* Функцию выбрать самостоятельно. Использовать цикл `for`. Таблицу напечатать "красиво", с заголовком, выравниванием по столбцам и аккуратным форматированием чисел.

Задание 7.6. *Напечатать таблицу значений выражения $(x^2-p^2)/(x-p)-p$ для некоторого p и для x из отрезка $[a,b]$ с шагом h. Обратить внимание, что математически данное выражение равно x. Рассмотреть случаи, когда x и p отличаются на большое количество порядков (5, 10, 15, 20 и т.п.). Всегда ли в результате вычисления получается x? Как объяснить именно такие результаты?*

§2.3. Перечисления

Перечисляемый тип — тип данных, множество значений которого есть конечный набор идентификаторов. Это особый тип данных, создаваемый программистом: множество допустимых значений определяется в программе, значения этого типа можно сравнивать между собой. В Си перечисляемый тип сводится к целочисленному, а идентификаторы, которые могут использоваться в качестве значения типа становятся целочисленными константами (тип `int`). Например,

```
enum day {SUN, MON, TUE, WED, THU, FRI, SAT};

enum day d = MON;
```

Такая конструкция, во-первых, вводит тип данных, при этом собственно именем типа является именно сочетание `enum` и идентификатора (в данном примере `enum day`, а не просто `day`), что требует указание именно такого сочетания при объявлении переменных данного типа. Во-вторых, вводятся константы данного типа, по факту целочисленные, по умолчанию — все различные, нумерация начинается с нуля. Так как перечисления являются целочисленными константами их можно использовать в операторе выбора:

```
switch (d)
{
    case SUN: printf ("Sunday\n");   break;
    case MON: printf ("Monday\n");   break;
    /* ... */
    case SAT: printf ("Saturday\n"); break;
    default:  printf ("Invalid␣value\n");
};
```

Указание метки `default` обычно необязательно, так как других значений (если не присваивать переменной целочисленные значения напрямую) не возникает, но желательно, а вот перебрать все варианты необходимо.

Часть II. Процедурно-модульное программирование

Глава 3. Функции

§3.1. Синтаксис написания функции

Код функции в Си состоит из **заголовка** и **тела функции**. Заголовок представляет собой следующую конструкцию:

<тип возвр. значения> <имя функции> (<тип пар-ра 1> <имя пар-ра 1>, <тип пар-ра 2> <имя пар-ра 2>, ...) >

Параметров может не быть, тогда скобки должны быть пустыми. Возвращаемого значения может не быть, тогда в качестве его типа нужно указать ключевое слово `void`. Такие функции называются void-функции.

После заголовка в фигурных скобках следует тело функции. Оно представляет собой составной оператор, по сути блок операторов, список операторов, заключенных в фигурные скобки. В начале блока можно объявить **локальные** переменные, то есть переменные, которые будут доступны внутри функции и только внутри ее.

Возврат значения осуществляется с помощью оператора `return`. Оператор не только возвращает значение, но и прерывает работу функции. В void-функции можно использовать `return` без параметров, чтобы прервать ее работу.

Примеры.

1. Функция, возвращающая квадрат целого числа.

```
int sqr (int x)
{
    return x*x;
}
```

   Замечание. Следуя стилю стандартной библиотеки, функцию, возвращающую квадрат числа типа `double` следовало бы назвать `fsqr`.

2. Функция, вычисляющая целую степень числа (наивным, неэффективным образом).

```
int intpow (double x, unsigned n)
{
    double r = x;     /* локальная переменная для подсчета результата  */

    if (n == 0)       /* возведение в нулевую степень                  */
        return 1;     /* сразу возвращаем 1 и завершаем работу функции */

    for (; n>1; --n)  /* необходимо проделать n-1 умножение            */
        r *= x;

    return r;
}
```

   В Си все параметры в функции передаются по значению, то есть копируются, поэтому изменение формального параметра `n` внутри функции никак не скажется на значении фактического параметра в том месте, где функция была вызвана.

3. Функция, проверяющая, верно ли, что указанное целое число является точным квадратом.

```
int is_square (unsigned n)
{
    unsigned i = 0;
    while (i * i < n) i++; /* пока потенциальный корень не превысит n */

    /* здесь два варианта, либо i^2=n, те.. n --- точный квадрат
     * либо i^2>n, тогда n не является точным квадратом
     */

    return i * i == n;
}
```

Это логическая функция, она возвращает логическое значение (0 или 1). Соответственно, проверка с ее помощью будет выглядеть как `if (is_square(k))` иди `if (!is_square(k))` — пример ситуации, когда использование `!=0` или `==0` не только создаст дополнительные ненужные действия, но и ухудшит воспринимаемость программы, выведет на первый план детали реализации (представление логических данных в Си), а не суть.

Обратите внимание, что в последней строке было бы избыточно писать

```
if (i * i == n)
    return 1;
else
    return 0;
```

или

```
return i * i == n ? 1 : 0;
```

так как фактически требуется именно значение указанного в функции выражения (0 или 1), а указанные конструкции не вычисляют новой информации.

§3.2. Методика написания функции

Разбиение программы на функции преследует главным образом следующие цели:

- повторное использование кода (вместо того, чтобы писать код по новой или копировать его, достаточно вызвать функцию);
- исключение повторяющихся участков кода (в повторяющихся участках кода может возникнуть повторяющаяся ошибка, ее технически трудно исправлять, аналогичная проблема возникает при необходимости модифицировать или улучшить данный код);
- уменьшение количества ошибок и облегчение их поиска (легче правильно написать, осознать, протестировать, отладить и проверить короткий код каждой функции отдельно, чем всей неразделенной на части программы целиком);
- разделение труда программистов (разные функции одной программы могут создаваться разными людьми);
- сокрытие деталей реализации и повышение уровня абстракции (для работы с функцией достаточно знать, как она вызывается и что делает, но не важно как именно она это делает — всякая функция есть черный ящик);
- облегчение сопровождения программ (модификация деталей реализации — внутренности черного ящика —- при сохранении внешности черного ящика не должна сказываться на работоспособности остальных функций).

При написании функций следует учитывать эти цели. Для того, чтобы качественно написать функцию, необходимо следовать ряду рекомендаций и соглашений.

- Каждая функция должна решать какую-то свою задачу. Если это расчетная функция, то она должна производить вычисление, если интерфейсная — отвечать за ввод и вывод и т.п. Смысл разделения программы на функции именно в том, чтобы разделить большую задачу на малые подзадачи.

  Строгое разделение интерфейсной и рабочей части вообще важный момент в программировании, так как это позволяет модифицировать интерфейс, не затрагивая рабочие функции.

- Прежде чем написать функцию, следует четко сформулировать, что именно она делает, какие у нее входные и выходные данные (внешняя сторона черного ящика). Название функции и параметров должно говорить само за себя.

  Всякая функция должна снабжаться комментарием, поясняющим, что она делает, каков смысл каждого из ее параметров, что означает возвращаемое значение.

  В идеале функция должна быть универсальной и библиотечного качества (если, конечно, это возможно, т.е. она не чисто вспомогательная), то есть быть приспособленной для обработки любых данных в любых условиях, в том числе чтобы могла быть использована и в другой программе, под это и следует затачивать ее прототип и описание.

- Следует отказаться от использования глобальных переменных.

  Не следует путать локальные переменные и формальные параметры функции: у них принципиально различная задача — первые нужны для хранения данных внутри функции (промежуточных вычислений и т.п.), вторые — для обмена данными между функциями (входных и выходных данных алгоритма).

Лабораторная работа №8

Функции.

В данной лабораторной главная задача — написание функции, то есть рабочей части программы. Однако также требуется написание и интерфейсной части, которая позволит проиллюстрировать и протестировать работу функции.

Задание 8.1. *Написать функцию, находящую цифровой корень DR натурального числа n в системе счисления p.*

$$DR_p(n) = \begin{cases} n, & n < p, \\ DR_p(S_p(n)), & n \geqslant p, \end{cases}$$

где $S_p(n)$ — сумма p-ичных цифр числа n. Рекурсию не использовать.

Задание 8.2. *Напишите функцию, которая определяет является ли первый принимаемый параметр некоторой натуральной степенью второго параметра (для (n, m) указать верно ли, что есть некоторый $k \in \mathbb{N}$, такой что $n = m^k$, числа n и m целые.)*

Задание 8.3. *Написать функцию*

(a) которая принимает натуральное число n и выводит на экран квадрат из звездочек со стороной n;

(b) с тремя целыми параметрами m, n, d, изображающую в текстовом окне терминала при помощи символов * рамку размером m на n толщиной d;

Задание 8.4. *Написать функцию, которая определяет количество разрядов введенного целого числа.*

Задание 8.5. *Написать функцию приближенного вычисления квадратного корня из числа a с помощью метода итераций*

$$x_0 = a,$$
$$x_n = \frac{1}{2}\left(x_{n-1} + \frac{a}{x_{n-1}}\right).$$

с некоторой относительной погрешностью ε (итерации прекратить когда $|(x_n - x_{n-1})/x_n| < \varepsilon$). Учесть, что аргументом функции может оказаться отрицательное число (результат — не-число), положительная бесконечность (результат — она же). Замечание. Цикл `do..while` позволит написать даст более короткий код.

Задание 8.6. *Написать функцию, вычисляющую приближенное значение следующих функций, используя указанный степенной ряд.* Бесконечную сумму заменить конечной, остановившись на первом слагаемом, которое окажется меньше (по модулю, если необходимо) величины ε. Прямое вычисление степени и факториала исключить, заменив нахождение значения каждого последующего слагаемого через предыдущее домножением на соответствующий коэффициент. Параметрами функции должны быть аргумент x и число ε.

(a) $\exp x = \sum\limits_{n=0}^{\infty} \dfrac{x^n}{n!}$;

(b) $\sin x = \sum\limits_{n=0}^{\infty} \dfrac{(-1)^n x^{2n+1}}{(2n+1)!}$;

(c) $\cos x = \sum\limits_{n=0}^{\infty} \dfrac{(-1)^n x^{2n}}{(2n)!}$;

(d) $\ln(1+x) = \sum\limits_{n=1}^{\infty} \dfrac{(-1)^{n-1} x^n}{n}$;

(e) $\operatorname{sh} x = \sum\limits_{n=0}^{\infty} \dfrac{x^{2n+1}}{(2n+1)!}$;

(f) $\operatorname{ch} x = \sum\limits_{n=0}^{\infty} \dfrac{x^{2n}}{(2n)!}$;

§3.3. Параметр-указатель как выходной параметр

Синтаксически у функции может быть только одно возвращаемое значение. Однако иногда возникают ситуации, когда выходные данные алгоритма предполагают возврат более, чем одного значения. Это возможно

при использовании **указателей**: если в функцию передать адрес переменной, то, используя этот адрес, можно изменить значение данной переменной: хотя сам адрес будет скопирован, это будет все тот же номер ячейки памяти.

Указатели объявляются с помощью **символа указателя** `*` перед именем переменной или параметра. Например, `double *x` — указатель на переменную типа `double`, `int *n` — указатель на переменную типа `int`. Тип указателя важен, чтобы Си знал, значение какого типа содержится в указываемой ячейке памяти.

Получение указателя из переменной осуществляется с помощью операции взятия адреса `&`. Получение данных по указателю — с помощью унарной префиксной операции разыменования `*`. Эти данные можно как прочитать, так и изменить, то есть в примерах выше с `*x` и `*n` можно обращаться как с обычной переменной.

Следующая функция возвращает и остаток и неполное частное двух целых чисел. При этом, функция возвращает 0, если деление произведено успешно и -1 в случае ошибки.

Это один из стандартных способов проинформировать о проблеме, возникшей внутри функции: возвращаемое значение есть не более чем сигнал о успехе или ошибке в ее работе, в то время как сами выходные данные передаются через параметр-указатель.

```
/* Вычисление остатка и неполного частного
 * входные параметры:  n   - делимое, m - делитель
 * выходные параметры: div - неполное частное, mod - остаток
 * возвращает 0 в случае успеха, -1 в случае ошибки деление( на 0)
 */
int divmod (int n, int m, int *div, int *mod) /* указатели - выходные параметры */
{
    if (!m) return -1;

    *div = n / m; /* присваивание значения разыменованным указателям */
    *mod = n % m; /* как в обычные переменные                        */
    return 0;
}
```

Использовать данную функцию можно, например, так

```
int i, j, d, m;
/* ... */
if (divmod(i, j, &d, &m))  /* взяты адреса переменных для записи результата */
    printf("Ошибка, деление на ноль\n");
else
    printf("%d / %d = %d, %d %% %d = %d\n", i, j, d, i, j, m);
```

Функцию можно усовершенствовать: в примере выше если один из указателей `NULL`, выполнение функции приведет к краху программы. Это можно избежать вернув -1 при получении нулевого указателя. Однако лучше предоставить пользователю функции возможность корректно отказаться от ненужного при конкретном вызове выходного параметра — передать `NULL`.

```
/* Вычисление остатка и неполного частного
 * входные параметры:  n   - делимое, m - делитель
 *                     ( могут быть NULL, если не нужны )
 * выходные параметры: div - неполное частное, mod - остаток
 * возвращает 0 в случае успеха, -1 в случае ошибки деление( на 0)
 */
int divmod (int n, int m, int *div, int *mod)
{
    if (!m) return -1;

    if (div) *div = n / m; /* проверяется на 0 именно указатель */
    if (mod) *mod = n % m; /* писать mod != NULL излишне        */
    return 0;
}
```

Лабораторная работа №9

Функции и параметры-указатели

Задание 9.1. *Написать функцию, одновременно вычисляющую сумму и число p-ичных цифр целого числа.* Так

как данная функция не может завершиться с ошибкой, одно из выходных чисел можно сделать возвращаемым значением.

Задание 9.2. *Написать функцию, которая упорядочивает значение двух целочисленных переменных: параметрами являются адреса двух переменных a и b, если $a > b$, функция меняет их значения местами.* В данном случае параметры-указатели будут одновременно и входными и выходными данными.

Задание 9.3. *Написать функцию, которая сортирует значения трех вещественных переменных.*

Задание 9.4. *Написать функцию анализа квадратного уравнения $ax^2+bx+c=0$. Функция должна возвращать целочисленную константу-перечисление, информирующую о полученном случае (два вещественных корня, один кратный корень, два комплексных корня, линейное уравнение с одним корнем ($a=0$, $b\neq 0$), уравнение, не имеющее корней ($a=0$, $b=0$, $c\neq 0$), любое вещественное число является корнем ($a=b=c=0$), среди коэффициентов есть бесконечность или не-число). Корни (если есть) должны возвращаться через дополнительные параметры-указатели (учесть, что может быть передано* `NULL`*, тогда корень вычислять и возвращаться не надо). В интерфейсной части программы проанализировать возвращаемое функцией перечисление с помощью* `switch`*, вывести случай и корни (если есть).*

Замечание. Перечисляемый тип излагается в лекционном курсе.

Глава 4. Модули

§4.1. Разделение программы на модули в Си

На практике программы бывают достаточно большие, состоящие из значительных по объему разделов и подзадач, решающихся с помощью набора взаимосвязанных функций, многие из которых являются чисто вспомогательными. Непрактично помещать весь исходный код программы в один файл уже хотя бы потому, что данный файл трудно будет осознать, в нем будут присутствовать разнородные и никак не связанные между собой группы функций. Кроме того, если программа создается группой разработчиков, трудно организовать одновременную их работу с единственным файлом. Также при отладке программы при малейшем изменении файла требуется его полная компиляция, что может занять значительное время.

Использование модульного программирования решает следующие задачи:

- повторное использование кода (вместо того, чтобы копировать код функции из другой программы достаточно подключить модуль);
- разделение труда программистов (разные модули одной программы могут создаваться разными людьми);
- сокрытие деталей реализации и повышение уровня абстракции (для работы с модулем достаточно знать его интерфейсную часть, то есть какие возможности он предоставляет, но не важно как именно он это делает — модуль расширяет понятие черного ящика до группы функций, отвечающих за некоторый круг задач или подзадачу);
- облегчение сопровождения программ (модификация деталей реализации — внутренности черного ящика — - при сохранении интерфейса — внешности черного ящика — не должна сказываться на работоспособности остальных модулей).

В Си модули как элемент программы не поддерживаются, концепцию модульного программирования можно реализовать при помощь **раздельной компиляции**.

- Программа разбивается на несколько файлов исходного кода (с расширением `.c`), содержащих реализацию групп функций, решающих выбранную подзадачу — реализация модуля.
- Для того, чтобы функции, реализованные в различных `.c`-файлах могли вызывать друг друга, создаются заголовочные файлы (с расширением `.h`), содержащие прототипы функций и подключаемые по мере необходиости — интерфейсная часть модуля.
- Каждый `.c`-файл компилируется отдельно в объектый модуль (с расширением `.o`), затем объектные модули компануются в исполняемый файл программы.
- При сборке проекта IDE компилирует только те `.c`-файлы которые были изменены сами или подключают измененные `.h`-файлы (обычно решается при помощи сравнения времени создания объектного модуля и рассматриваемых файлов исходного кода и заголовочных файлов.

§4.2. Создание заголовочных файлов

При создании заголовочных файлов следует руководствоваться рядом требований и рекомендаций.

- Следует выбирать говорящее имя заголовочного файла, соответствующее тому, какую задачу решает данный модуль.
- Всякий заголовочный файл должен быть защищен от двойного подключения, в противном случае может возникнуть ошибка компиляции из-за двойного определения одних и тех же идентификаторов. Делается это с помощью комбинации, подобной

```
#ifndef FILE_H_INCLUDED
#define FILE_H_INCLUDED

/* Здесь код заголовочного файла */

#endif /* FILE_H_INCLUDED */
```

  Константа `FILE_H_INCLUDED` может быть любой, но должна быть уникальной среди всех заголовочных файлов, обычно она формируется из имени этого файла.
- В заголовочном файле должны присутствовать только объявления: введение типов данных, прототипы функций, символические константы и т.п. Никаких определений, в том числе кода функций и тем более глобальных переменных там быть не должно, это вызовет ошибку двойного определения при компоновке.
- Всякое объявление в заголовочном файле следует снабжать комментарием, что именно означает или делает вводимый идентификатор или константа. Комментарий к функции следует помещать именно в заголовочный файл. Также заголовочный файл (как и файлы исходного кода) следует снабжать общим комментарием, поясняющим, какая группа задач или подзадача решается с помощью данного модуля и кто его автор. По сути комментарий в заголовочном файле — почти готовая документация к модулю.
- Совершенно необязательно, чтобы все функции из файла исходного кода, имели прототип в заголовочном файле: часть из них может быть внутренними, представлять собой детали реализации, и не быть вызываемой из вне модуля. Объявления таких функций следует предварять ключевым словом `static`.

Лабораторная работа №10

Модули

В данной лабораторной главная задача — написание модуля (заголовочного файла и файла исходного кода), то есть рабочей части программы. Однако также требуется написание и интерфейсной части, которая позволит проиллюстрировать и протестировать работу модуля.

Задание 10.1. *Написать модуль анализа свойств целых неотрицательных чисел. Интерфейс модуля должен предоставлять функции*

(a) вычисления суммы p-ичных цифр числа;

(b) вычисления количества p-ичных цифр числа;

(c) вычисления цифрового корня числа в системе счисления p;

(d) определения, делится ли число на сумму своих p-ичных цифр;

(e) определения, делится ли число на каждую из своих p-ичных цифр;

(f) определения, делится ли число на каждую из своих четных p-ичных цифр;

(g) определения, делится ли число на каждую из своих нечетных p-ичных цифр;

(h) определения, делится ли число на каждую из своих нечетных p-ичных цифр;

(i) определения, делится ли число на наименьшую из своих p-ичных цифр;

(j) определения, делится ли число на наибольшую из своих p-ичных цифр;

(k) определения, является ли число палиндромом.

(l) определения, является ли число n простым (перебором делителей до $\lfloor\sqrt{n}\rfloor$).

Задание 10.2. *Написать модуль анализа квадратного уравнения $ax^2+bx+c=0$. В заголовочном файле модуля следует определить перечисляемый тип, информирующий о результате анализа, (два вещественных корня, один кратный корень, два комплексных корня, линейное уравнение с одним*

корнем ($a = 0$, $b \neq 0$), уравнение, не имеющее корней ($a = 0$, $b = 0$, $c \neq 0$), любое вещественное число является корнем ($a = b = c = 0$), среди коэффициентов есть бесконечность или не-число), и функция анализа, возвращающая соответствующую константу-перечисление, а также корни (если есть) через дополнительные параметры-указатели (учесть, что может быть передано NULL, тогда корень вычислять и возвращаться не надо). В интерфейсной части программы проанализировать возвращаемое функцией перечисление с помощью `switch`*, вывести случай и корни (если есть).*

Глава 5. Рекурсия

Функция в Си может вызывать сама себя как напрямую (прямая рекурсия), так и опосредованно (непрямая рекурсия). Это достигается за счет того, что для каждого вызова функции создается полный экземпляр всех локальных переменных и параметров данной функции, поэтому различные вызовы одной и той же функции между собой не связаны.

При написании рекурсивных функций следует учитывать ряд замечаний и рекомендаций.

- Часто код рекурсивной функции проще и понятнее, однако нередко менее эффективен. Отдавать предпочтение рекурсии следует только в том случае, когда затраты труда программиста по разработке итеративного алгоритма заведомо не окупятся эффективностью этого алгоритма.
- При написании рекурсивной функции следует проследить, чтобы был реализован **выход из рекурсии**: при некоторых значениях параметров функция должна завершиться без вызова себя, причем к этому значению параметров рекурсия должна прийти обязательно.

  Данные функций хранятся в т.н. **стеке** программы, размер которого ограничен операционной системой, поэтому бесконечная рекурсия приведет к ошибке переполнения стека, а не к бесконечному циклу.

Лабораторная работа №11
Рекурсия

Задание 11.1. *Написать функцию, вычисляющую*

(a) сумму цифр числа (только рекурсивно);

(b) цифровой корень числа (только рекурсивно);

(c) наибольший общий делитель (с помощью алгоритма Евклида) — рекурсивно и итеративно;

(d) наименьшее общее кратное с использованием предыдущей функции и соотношения

$$\text{НОК}(n, m) \cdot \text{НОД}(n, m) = n \cdot m.$$

Задание 11.2. *Написать рекурсивные и итеративные функции, вычисляющую значение в точке x многочлена специального вида порядка n, определяемого рекуррентным соотношением:*

(a) многочлен Чебышёва 1-го рода

$$T_0(x) = 1, \quad T_1(x) = x,$$

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x).$$

(b) многочлен Чебышёва 2-го рода

$$U_0(x) = 1, \quad U_1(x) = 2x,$$

$$U_{n+1}(x) = 2xU_n(x) - U_{n-1}(x).$$

(c) многочлен Лежандра

$$P_0(x) = 1, \quad P_1(x) = x,$$

$$P_{n+1}(x) = \frac{2n+1}{n+1}\, x\, P_n(x) - \frac{n}{n+1}\, P_{n-1}(x).$$

(d) многочлен Лаггера

$$L_0(x) = 1, \quad L_1(x) = 1 - x,$$

$$L_{n+1}(x) = \frac{1}{n+1}\left((2n + 1 - x)L_n(x) - nL_{n-1}(x)\right).$$

(e) многочлен Падована

$$P_1(x) = 1, \quad P_2(x) = 0, \quad P_3 = x,$$

$$P_n(x) = xP_{n-2}(x) + P_{n-3}(x).$$

(f) многочлен Фибоначчи

$$F_0(x) = 0, \quad F_1(x) = 1,$$

$$F_n(x) = xF_{n-1}(x) + F_{n-2}(x).$$

(g) многочлен Эрмита

$$H_0(x) = 1, \quad H_1(x) = x,$$

$$H_{n+1}(x) = xH_n(x) - nH_{n-1}(x).$$

Задание 11.3. *Написать рекурсивную функцию, которая печатает в обратном порядке цифры p-ичной ($p \leqslant 16$) записи целого неотрицательного числа.*

Задание 11.4. *С клавиатуры вводится последовательность ненулевых вещественных чисел. Заранее количество вводимых чисел неизвестно, признак конца ввода — 0 (в последовательность не входит). Написать рекурсивную функцию (не используя цикл), которая находит*

(a) сумму этих чисел;

(b) наибольшее из этих чисел;

(c) наименьшее из этих чисел;

Указание. Функция, предоставляемая интерфейсом модуля, должна быть без параметров и вызывать требуемую рекурсивную функцию, доступную только в реализации модуля.

Часть III. Структуры данных

Глава 6. Массивы: индексный подход

§6.1. Статические массивы

Массив — набор конечного числа данных одного типа, называемых **элементами массива**, расположенных в памяти непрерывно. Номер элемента массива называется **индексом**. **Статические массивы** — массивы, размер (количество элементов) которых определяется на стадии компиляции (написания) программы и не может изменяться в процессе ее исполнения.

Объявление статических массивов в Си проводится следующим образом:

⟨*тип*⟩ ⟨*идентификатор*⟩[⟨*целочисленная константа*⟩],

например

```
float a[10];
int p[3];
```

Объявления однотипных массивов могут чередоваться с объявлениями обычных переменных того же типа и между собой. Многомерные массивы представляют собой массивы массивов, их объявления выглядят, например, так:

```
float m[10][5];
int v[3][2][10];
```

Доступ к элементам массива осуществляется по индексу, нумерация элементов массива осуществляется с нуля:

```
a[0] = 5;
x = m[2][3];
```

Часто разумно объявлять размер массива-буфера с помощью символической константы, тогда все функции, работающие с данным массивом, будут иметь возможность проверить выход за пределы буфера.

```
#define ARRAY_SIZE 100
/* ... */
    double h[ARRAY_SIZE], w[ARRAY_SIZE];
/* ... */
    if (asize >= ARRAY_SIZE) return array_error_code;
```

Си не инициализирует элемента массива автоматически, поэтому их использование без инициализации — то же самое, что использование неинициализированных переменных. Кроме того, при работе с массивами Си не проверяет выход индекса за пределы размера массива, чтение элементов вне массива приводит к появлению непредсказуемого значения, запись — к порче данных. Проверку следует делать вручную, но только в тех случаях, когда трата времени на проверку оправдана:

```
if (asize >= ARRAY_SIZE) return array_error_code;
for (i = 0; i < asize; ++i)
    a[i] = create_array_value(i);
```

Первой строкой возможность выхода `i` за пределы массива исключена, поэтому внутри цикла такая проверка нетребуется.

Следует обратить внимание на условие выхода из цикла в предыдущем примере: это типичное применение цикла со счетчиком для прохода массива в Си. Использование

```
for (i = 0; i <= asize-1; ++i)
```

неразумно в силу дополнительной операции, а

```
for (i = 0; i != asize; ++i)
```

чревато появлением ошибки в случае, если цикл изменится на

```
for (i = 0; i != asize; i+=2)
```

(для нечетных `asize` условие выхода из цикла не будет достигнуто).

В Си статические массивы, объявленные внутри функции, располагаются в стеке программы. Статические массивы имеют одно преимущество — их создание (отведение места в памяти) осуществляется быстро — и ряд недостатков. Во-первых, их размер нельзя определить исходя из входных данных программы, поэтому если он не фиксирован (например, в массиве для хранения координат точки в пространстве заранее известно, что в нем будет три элемента, в этом случае разумно использовать статический массив), то приходится отводить число элементов, бóльшее, чем типично используемое, то есть создавать некоторый **буфер** данных, участок памяти, который будет использоваться частично. В этом случае в процессе исполнения программы должна проводиться проверка на то, что реально используемый объем данных меньше размера буфера. Во-вторых, в силу размещения массива в стеке его размер не должен быть слишком большим, чтобы не вызвать переполнение стека.

§6.2. Указатель как массив

Для того, чтобы вычислить адрес элемента массива необходимо и достаточно знать тип элементов массива и адрес первого (нулевого элемента). Всю эту информацию сообщает типизированный указатель. Поэтому указатель может использоваться как одномерный массив. Например, для массива

```
int a[5];
```

совместимым указателем будет указатель на `int`:

```
int *p;
/* ... */
p = a;
```

при таком присваивании использование `p` и `a` для получения элементов массива эквивалентно.

```
p[1] = 5;
a[2] = p[1];
```

Отличие массивов и указателей в том, что у них разный размер: `sizeof(a)` вернет $5 \cdot sizeof(int)$, а `sizeof(p)` вернет размер указателя, $sizeof(void*)$. Также массив — фактически постоянный адрес (не путать с указателем на константу), так же как и переменная — постоянный адрес данных в памяти, ему нельзя присвоить другое значение, чтобы он указывал на другие данные.

Для многомерных массивов совместимым указателем является также указатель на элемент массива — то есть на массив меньшей размерности, например

```
float m[2][3][4];
float (*b)[3][4];
b = m;
```

Скобки в третьей строке обязательны, так как в силу приоритета знака `*` объявление

```
float *b[3][4];
```

создаст двумерный массив указателей на `float`, а не указатель на двумерный массив элементов типа `float`.

§6.3. Массив как параметр функции

При написании функций обработки массивов следует учесть, что де-факто в качестве параметра функции может выступать только указатель, но не массив. Следующие прототипы эквивалентны.

```
void f (int a[10]);
void f (int a[1]);
void f (int a[]);
void f (int *a);
```

аналогично для многомерных массивов

```
void g (int a[10][15][2]);
void g (int a[1][15][2]);
void g (int a[][15][2]);
void g (int (*a)[15][2]);
```

Если нет причин указывать в качестве формального параметра функции именно синтаксическую единицу "массив" разумнее использовать указатель, как более стандартный подход. Массив не может быть возвращаемым значением функции.

Таким образом, во-первых, массив не копируется в область данных функции и может быть изменен внутри функции, если не передавать указатель на константу. Поэтому, если массив является чисто входным параметром функции его следует объявить как указатель на константу, если массив является выходным параметром — как простой указатель. Во-вторых, в функции нет информации о длине массива, ее следует передавать отдельным параметром, для этих целей стандартно используется тип данных `size_t`.

```
void foo (const int *in_array, size_t s);

void bar (int *in_out_array, size_t s);

void voo (const int *in_array, int *out_array, size_t s);
```

В последней функции предполагается, что выходной массив имеет ту же длину, что и входной массив.

Поскольку в реальных задачах часто бывает невозможно определить размер массива на стадии написания программы — он будет определен пользователем или входными данными программы при запуске — следует учесть, что при работе со статическими массивами следует создавать массивы с некоторым разумным "запасом", то есть с расчетом на то, что реальный массив будет меньше отведенного буфера. Слишком большой буфер, однако, использовать тоже не следует, так как это приведет к неоправданному расходу памяти.

Разумно установить размер буферов массивов с помощью символической константы для упрощения его изменения.

Для выходных параметров-массивов функций следует учитывать, что если формируется другой массив (отличный от входного) и он может оказаться бóльшей длины, чем исходный массив, то необходимо не только вернуть размер нового массива, но и проводить проверку на то, что буфер имеет достаточный размер, запросив этот размер в функцию, и вернув ошибку, если это не так, например, следующим образом:

```
/* Функция обрабатывает массив in_array размера in_array_size
 * в массив, записываемый в out_array, размер буфера которого out_array_size_limit.
 * Функция возвращает размер выходного массива или out_array_size_limit+1, если
 * выходной массив не может вместить все элементы.
 */
size_t f (const int *in_array,  size_t in_array_size,
                int *out_array, size_t out_array_size_limit
         );
```

Следует отметить, что функции, написанные с учетом этих обстоятельств, будут пригодны и для обработки динамических массивов, поэтому данный подход важен.

При работе с двумерными массивами следует учесть необходимость передачи в функцию соответствующего указателя:

```
#define ARRAY_BUFFER_SIZE 25

void foo (const double a[][ARRAY_BUFFER_SIZE], size_t w, size_t h);

/* объявление массива в( main): */
/* ... */

double a[ARRAY_BUFFER_SIZE][ARRAY_BUFFER_SIZE];
size_t w = /* фактическое число столбцов ширина() */,
       h = /* фактическое число строк высота() */;
```

Данный подход не может быть использован при работе с динамическими массивами, но рекомендуется к освоению.

Лабораторная работа №12

Массивы

Задание 12.1. *Реализовать функции ввода с клавиатуры и вывода на экран одномерных массивов целых и вещественных чисел заданной пользователем длины*

Задание 12.2. *Реализовать следующие функции анализа одномерных массивов.*

(a) Дан массив ненулевых целых чисел. Определить, сколько раз элементы массива при просмотре от его начала меняют знак.

(b) В массиве из n целочисленных элементов числа образуют неубывающую последовательность. Несколько элементов, идущих подряд, равны между собой. Сколько различных чисел имеется в массиве?

(c) Дан массив целых чисел. Рассмотреть отрезки массива (группы идущих подряд чисел), состоящие из нечетных (четных) чисел. Получить наибольшую из длин рассматриваемых отрезков.

(d) Найти в массиве вещественных чисел длину наибольшего участка упорядоченности (монотонности).

(e) Определить, является ли данный массив упорядоченным по возрастанию (убыванию).

(f) Найти второй по величине элемент массива (наибольший элемент массива при исключении максимального элемента этого массива).

Задание 12.3. *Реализовать следующие функции обработки одномерных массивов (входной массив должен измениться)*

(a) Удалить из массива целых чисел все четные (нечетные) элементы. (Под удалением элемента массива следует понимать исключение этого элемента из массива путем смещения всех следующих за ним элементов влево на одну позицию). Выполнить операцию за один проход массива, используя два индекса для чтения и записи.

(b) Удалить из массива все повторяющиеся элементы, оставив их первые вхождения, т. е. в массиве должны остаться только различные элементы. (Под удалением элемента массива следует понимать исключение этого элемента из массива путем смещения всех следующих за ним элементов влево на одну позицию).

(c) Написать функции сортировки массива тремя различными методами, хотя бы один из которых должен быть в среднем за время $O(n \log n)$, один — за $O(n^2)$. Проиллюстрировать работу данных функций путем подсчета числа произведенных перестановок и сравнений элементов массива на конкретных примерах.

Задание 12.4. *Реализовать следующие функции обработки одномерных массивов (исходный массив должен остаться неизменным).*

(a) После каждого элемента исходного целочисленного массива дописать его квадрат.

(b) Все нечетные элементы исходного целочисленного массива удалить, четные повторить.

Задание 12.5. *Реализовать следующие функции анализа и обработки двумерных массивов вещественных чисел.*

(a) Найти номер строки массива, сумма элементов в которой наибольшая.

(b) Отсортировать строки массива по возрастанию их наибольших (наименьших) элементов.

Глава 7. Работа с указателями и памятью

§7.1. Функциональный указатель

Нередки ситуации, когда параметром функции должна служить другая функция. Фактически требуется передать адрес функции куда нужно передать управление, соответственно, можно задать указатель, который будет указывать на данные *функционального типа* — **функциональный указатель**.

Общий принцип объявления указателя в Си состоит в превращении объявления простого идентификатора (в данном случае прототипа функции) в объявление указателя путем предварения имени объявляемого идентификатора символом указателя `*`. Однако, в силу приоритета знаков операций в Си, идентификатор и символ указателя следует заключить в скобки, чтобы звездочка относилась к идентификатору, а не типу возвращаемого значения.

Пусть есть функция

```
int foo (int, double);
```

тогда указатель на функцию такого типа будет объявляться как

```
int (*pfnc) (int, double);
```

что не следует путать с записью

```
int *foo2 (int, double);
```

объявляющую функцию foo2, которая в отличие от foo, возвращает `int *`, а не `int`.

Если же нужно объявить указатель, совместимый с прототипом foo2, то он будет записан как

```
int *(*pfnc2) (int, double);
```

Функциональному указателю можно присваивать адреса функций совместимого типа и вызывать их через указатель:

```
int (*pfnc) (int, double);

/* ... */

int foo (int, double);
int bar (int, double);
int fnc (int, double);

/* ... */

pfnc = &foo; /* или */ pfnc = &bar; /* или */ pfnc = &fnc;

int k = (*pfnc) (1, M_PI);
```

На самом деле фактически не существует отдельного “функционального типа”, поэтому каждая функция — не более, чем адрес кода, и результатом разыменования функционального указателя будет тот же функциональный указатель, поэтому взятие адреса и разыменования можно опускать:

```
pfnc = foo; /* или */ pfnc = bar; /* или */ pfnc = fnc;

int k = pfnc (1, M_PI);
```

Функциональный указатель можно использовать в качестве параметра функции:

```
void workfnc (int (*f) (int, double))
{
    /* ... */
    i = f (k, b);
}

/* ... */

workfnc(foo);
```

Рекомендуется использовать `typedef` для создания односложных псевдонимов для типа функциональный указатель. Правило использование `typedef`: объявление переменной, предваренное `typedef`, превращается в объявление псевдонима типа

```
typedef int (*tfnc) (int, double);

tfnc pfnc = foo;

/* или */

void workfnc (tfnc f)
{
    /* ... */
    i = f (k, b);
}
```

Лабораторная работа №13
Функциональный указатель

Задание 13.1. *Реализовать функции численного интегрирования указанным методом. Аргументом функции должны выступать границы промежутка, указатель на функцию (одного аргумента типа* `double` *с возвращаемым значением типа* `double` *и входного массива параметров* `double*`*) и число точек разбиения промежутка. Использовать метод*

(a) левых прямоугольников;

(b) правых прямоугольников;

(c) средних прямоугольников;

(d) трапеций;

(e) парабол.

Задание 13.2. *Реализовать функцию, которая с помощью функции 13.1 проводит интегрирование методом итераций, увеличивая число разбиений промежутка в 2 раза на каждом шаге до тех пор, пока два последовательных результата не окажутся отличающимися менее, чем на ε (относительная погрешность) Аргументом функции должны выступать границы промежутка, указатель на функцию (одного аргумента типа* `double` *с возвращаемым значением типа* `double` *и входного массива параметров* `double*`*), относительная погрешность и указатель на функцию численного интегрирования 13.1.*

Задание 13.3. *Протестировать функцию 13.2 на различных функциях и промежутках. Функции интегрирования, интегрируемые функции и интерфейс должны быть реализованы в отдельных единицах трансляции.*

§7.2. Арифметика указателей

Лабораторная работа №14
Обработка массивов с помощью арифметики указателей

Задание 14.1. *Реализовать следующие функции обработки и анализа одномерных массивов посредством арифметики указателей*

(a) ввода с клавиатуры и вывода на экран массивов целых и вещественных чисел заданной пользователем длины

(b) проверить, является ли заданный массив упорядоченным по возрастанию (убыванию);

(c) найти в массиве вещественных чисел длину наибольшего участка упорядоченности (монотонности);

(d) удалить из массива целых чисел все четные (нечетные) элементы (под удалением элемента массива следует понимать исключение этого элемента из массива путем смещения всех следующих за ним элементов влево на одну позицию), выполнить операцию за один проход массива, используя два указателя для чтения и записи.

Задание 14.2. *Найти в массиве "промахи измерений". Считаем, что в массиве записан ряд измерений некоторой величины, произведенных с погрешностью. Требуется выявить случаи измерений, погрешность которых слишком сильная (например, если измерение проводится одним и тем же инструментом, но разными людьми, можно выявить операторов, которые ошибаются слишком сильно, или, наоборот, при измерении одним и тем же человеком с помощью разных приборов, выявить прибор, выдающий плохие результаты; можно также исключить случайные ошибки — "промахи" — в серии измерений, выполненной на одном инструменте одним человеком, возникшие в результате ошибочных действий или влияния неучтенных факторов). Считаем, что измерение слишком ошибочно, если значение отклоняется от среднего более, чем на среднеквадратичное отклонение.*

(a) Найти среднее арифметическое $\bar{x} = 1/n \sum_{i=1}^{n} x_i$ и несмещенную оценку среднеквадратичного отклонения $s = 1/(n-1)\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}$ элементов массива $X = \{x_1, x_2, \ldots, x_n\}$.

(b) Найти указатель на первый элемент x' остатка массива, отклоняющегося от заданного значения a более, чем на ε (в смысле разности модулей значений), начиная с заданного элемента x (в функцию передается

указатель на один из элементов массива, начиная с которого проводится поиск вправо, если искомого элемента в остатке массива массиве нет, возвращается NULL).

(c) Вывести все элементы массива (индексы элементов), которые являются “промахом”.

Задание 14.3. *Реализовать следующие алгоритмы обработки упорядоченного массива данных любого типа посредством арифметики указателей. Массив передается в виде нетипизированного указателя и размера элемента в байтах. Также функция должна получать на вход функцию сравнения двух элементов данного типа (логическую как аналог операции больше/меньше или возвращающую -1, если первый аргумент больше второго, 0, если аргументы равны, 1, если второй аргумент больше), также передаваемых в нее как нетипизированные указатели.*

(a) Бинарный поиск — возвращает указатель на найденный элемент или `NULL`, если элемента в массиве нет.

(b) “Бинарное место” — возвращает индекс элемента, на который следует вставить искомый элемент в массив с сохранением упорядоченности при условии сдвига элементов, начиная с возвращенного. Возвращает 0, если элемент меньше всех элементов массива, количество элементов массива, если он больше всех. Если такой элемент в массиве уже есть (или их несколько), возвращает номер следующего за равными элемента (минимизация сдвига). Функция только находит индекс, сдвигать элементы не требуется.

§7.3. Динамические массивы

Лабораторная работа №15
Динамические одномерные массивы

Задание 15.1. *Обеспечить ввод динамического одномерного массива с вводом числа элементов, выделением памяти, и ввода-вывода элементов посредством функций из задачи 12.1.*

Задание 15.2. *Убедиться в применимости других функций лабораторной 12 и 14 к динамическим одномерным массивам.*

§7.4. Алгоритмы обработки массивов

Лабораторная работа №16
Алгоритмы обработки массивов

Задание 16.1. *Реализовать следующие алгоритмы обработки массивов целочисленных элементов:*

(a) сортировка выбором;

(b) сортировка пузырьком;

(c) сортировка вставками;

(d) сортировка слиянием;

(e) быстрая сортировка;

(f) слияние упорядоченных массивов;

(g) бинарный поиск итеративно (возвращает указатель на найденный элемент или NULL, если элемент не найден);

(h) бинарное место итеративно (возвращает указатель на элемент массива, перед которым следует вставить искомый элемент, чтобы сохранилось свойство упорядоченности, в т.ч. на “элемент, следующий за последим”, если вставка должна производиться в конец массива). При реализации алгоритма следует использовать арифметику указателей, но не индексацию.

Задание 16.2. *Сравнить различные алгоритмы сортировок для одного и того же массива, подсчитав*

(a) количество сравнений;

(b) количество перестановок. Информация должна возвращаться функцией сортировки с помощью двух опциональных выходных параметров.

Задание 16.3. *Реализовать следующие алгоритмы обработки массивов элементов любых типов данных*

(a) сортировка выбором;

(b) сортировка пузырьком;

(c) сортировка вставками;

(d) сортировка слиянием;

(e) быстрая сортировка;

(f) слияние упорядоченных массивов;

(g) бинарный поиск итеративно (возвращает указатель на найденный элемент или NULL, если элемент не найден);

(h) бинарное место итеративно (возвращает указатель на элемент массива, перед которым следует вставить искомый элемент, чтобы сохранилось свойство упорядоченности, в т.ч. на "элемент, следующий за последим", если вставка должна производиться в конец массива). Массив в указанные функции передается в виде указателя на блок памяти (`void *`), размера элемента в байтах и количества элементов, кроме того в функцию передается функциональный указатель на функцию сравнения данных типа, соответствующего типу элемента массива. При реализации алгоритма следует использовать арифметику указателей. Реализовать функции сравнения для символов, целых чисел, чисел с плавающей точкой, строк символов и протестировать требуемые функции для массивов элементов различных типов (использовать строки символов фиксированной предельной длины или указатели на динамически выделяемые строки; в последнем случае размером элемента массива будет указатель на блок памяти, а перестановка элементов будет соответствовать перестановке указателей без фактического перемещения данных).

§7.5. Строки символов

Лабораторная работа №17

Обработка строк символов как массивов символов

Во всех задачах лабораторной обрабатывать строку как массив символов, без использования функций стандартной библиотеки, используя арифметику указателей, но не индексацию.

Задание 17.1. *Удалить "лишние" пробелы строки. Лишними считаются повторяющиеся пробелы (при наличии двух и более пробелов подряд следует оставить один), пробелы вначале и в конце строки. Выполнить за один проход без создания дополнительной строки (использовать два указателя — для чтения и записи символов).*

Задание 17.2. *Написать функцию, выделяющую подстроку из строки в новый строковый буфер. Подстрока должна определяться двумя целыми числами, (n, m), где n — номер первого символа подстроки, m — длина подстроки. Проследить, чтобы функция в любом случае вернула корректный результат (если конец подстроки выходит за границы строки вернуть допустимую часть, если начало подстроки лежит после конца строки вернуть пустую строку).*

Задание 17.3. *Написать функцию, выделяющую подстроку из строки в новый строковый буфер. Подстрока должна определяться двумя целыми числами, (n, m), где n — номер первого символа подстроки, причем если $n < 0$, то отсчитываемый с конца строки ($n = -1$ соответствует последнему символу), m — длина подстроки, причем если $m < 0$, то n становится индексом не первого, а последнего символа подстроки. Свести задачу к предыдущей путем пересчета значений m и n. Проследить, чтобы функция в любом случае вернула корректный результат (если конец или начало подстроки выходит за границы строки вернуть допустимую часть, если начало подстроки лежит после конца строки или конец подстроки — до начала строки вернуть пустую строку).*

Задание 17.4. *Проверить, является ли строка предложением-палиндромом (последовательность букв, но не знаков препинания и пробелов одинакова в обоих направлениях). Буквы разного регистра считаются разными.*

Задание 17.5. *Написать функцию получения строкового представления заданного целого числа в p-ичной системе счисления ($2 < p < 36$).*

Задание 17.6. *Написать функцию получения целого числа из его строкового представления в p-ичной системе счисления ($2 < p < 36$).*

Задание 17.7. *Подсчитать количество слов в строке*

(a) начинающихся на заданную букву;

(b) заканчивающихся на заданную букву;

(c) начинающихся и заканчивающихся одной и той же буквой;

(d) содержащих заданную букву не менее одного раза; (слова разделяются пробелами и знаками препинания, используются буквы латинского алфавита, буквы различного регистра считаются различными).

Задание 17.8. *Преобразовать строку*

(a) удалив последнюю букву слова;

(b) поменяв местами первую и последнюю букву каждого слова;

(c) удалив слова, в которых первая буква слова входит в это слово еще раз;

(d) удалив слова, в которых первая буква слова входит в это слово только один раз;

(e) удалив слова, в которых первая буква слова входит в это слово ровно два раза;

(f) удалив слова, длина которых максимальна;

(g) удалив слова, длина которых минимальна;

(h) удалив слова, в которых нет повторяющихся букв;

(i) удалив слова, каждая буква которых входит в это слово не менее двух раз;

(j) удалив из слова все последующие вхождения первой буквы этого слова;

(k) удалив из слова все предыдущие вхождения последней буквы этого слова;

(l) оставить в слове только первые вхождения каждой буквы;

(m) удалив среднюю букву в словах нечетной длины;

(n) вставив заданную букву в середину слов четной длины;

(o) перевернув каждое слово;

(p) удалив слова, первая и последняя буква которых совпадают;

(q) удвоив вхождение каждой буквы слова;

(r) удалив слова, если каждая буква входит в слово ровно два раза;

(s) удалив слова, если последняя буква входит в слово не более двух раз;

(t) оставить в каждом слове только последние вхождения каждой буквы;

(u) удалив одно вхождение каждой удвоенной буквы;

(v) оставив в каждом слове только последнее вхождение первой буквы;

(w) оставить в каждом слове только первое вхождение последней буквы;

(x) оставив только симметричные слова (палиндромы);

(y) оставив только те слова, буквы в которых упорядочены по алфавиту; (слова разделяются пробелами и знаками препинания, используются буквы латинского алфавита, буквы различного регистра считаются различными).

Лабораторная работа №18

Обработка строк символов посредством стандартной библиотеки

Задание 18.1. *Дана строка слов символов, слова разделяются пробелами и знаками препинания. Разделители могут повторяться (например, два и более пробелов подряд) или встречаться несколько разделителей подряд (например, запятая и пробел), могут встречаться разделители в начале и в конце строки. Преобразовать строку в массив слов. Для этого создать копию строки в памяти, в ней используя функцию* `strtok` *находить конец очередного слова, заменяя разделитель на символ с кодом 0, сохранять в массиве указателей на* `char` *начало очередного слова.*

Задание 18.2. *Дана строка слов символов, слова разделяются пробелами и знаками препинания. Найти*

(a) Число различных слов в строке.

(b) Число появлений и частоту встречаемости (процент от общего числа различных слов) каждого слова.

(c) Алфавитно-упорядоченный словарь всех слов в строке (без повторений).

(d) Словарь всех слов, упорядоченных по длине. Для простоты считать, что строка содержит слова, записанные латинскими буквами в одном регистре.

Указание. Для разбиения использовать функцию `strtok`. Для представления получаемых данных можно использовать структуры и массив структур.

Часть IV. Элементы графики

Средства вывода изображений на экран монитора предоставляются для самостоятельного изучения.

Лабораторная работа №19
Изучение графической библиотеки

Задание 19.1. *Изучите основные возможности выбранной графической библиотеки и способы подключения библиотеки к программе на стадии компоновки посредством IDE и опций компоновщика.*

Задание 19.2. *Изучите и протестируйте следующие возможности выбранной графической библиотеки:*

(a) создание окна графического вывода и закрытия его (перехода в графический режим и возврат к текстовому);

(b) определение размеров графического изображения;

(c) отображения точек, отрезков прямых линий, окружностей, эллипсов, секторов, прямоугольников и других геометрических фигур различным цветом и стилем;

(d) закраска областей и всего экрана различным цветом и стилем;

(e) вывод текста в изображение;

(f) использование абсолютного (по координатам окна) и относительного (перемещение пера) позиционирования точек;

(g) клавиатурный ввод в графическом окне.

Лабораторная работа №20
Построение графиков функций

Задание 20.1. *Используя графическую библиотеку напишите функцию, строящую отрезками по точкам с заданным шагом график функции (параметр–функциональный указатель) в заданном масштабе по осям (график должен оптимально располагаться на экране)*

(a) промежуток по оси абсцисс и ординат задается;

(b) промежуток по оси абсцисс задается, промежуток по оси ординат вычисляется как минимум и максимум значений функции. Оси и начало координат должны быть подписаны и снабжены засечками с некоторым “красивым” шагом.

Задание 20.2. *Снабдить программу интерфейсом изменения масштаба по осям с некоторым относительным шагом при нажатии определенных клавиш .*

Лабораторная работа №21
Построение фрактала

Задание 21.1. *Отобразить конечное приближение к следующему фрактальному множеству:*

(a) ковер Серпинского,

(b) треугольник Серпинского,

(c) кривую Коха, ограничив глубину рекурсии некоторым значением. Обеспечить пользователю возможность с помощью нажатия заданных клавиш регулировать глубину рекурсии.

Часть V. Данные и структуры данных

Глава 8. Файловый ввод и вывод

§8.1. Текстовые файлы

Лабораторная работа №22
Обработка текстовых файлов

Задание 22.1. *Дан текстовый файл, в котором записаны целые числа (в каждой строке разное количество). Определить*

(a) номер строки, в которой находится наименьшее (наибольшее) число;

(b) номер строки, в которой находится число, наиболее близкое к среднему арифметическому всех чисел файла;

При наличии нескольких строк, удовлетворяющих условию, указать первую из них. Делать предположение о длине наибольшей строки, о предельном количестве чисел в строке и предельном количестве строк в файле недопускается.

Задание 22.2. *Дан текстовый файл, в котором записаны вещественные числа (в каждой строке разное количество).*

(a) Сформировать новый текстовый файл, в который вывести средние арифметические и среднеквадратичные отклонения значений по каждой строке файла раздельно.

(b) Сформировать новый текстовый файл, в который вывести только те значения, которые отличаются от среднего арифметического значений строки не более чем на среднеквадратичные отклонения значений по каждой строке файла раздельно.

(c) Сформировать новый текстовый файл, в который вывести только те значения, которые отличаются от среднего арифметического значений строки более чем на среднеквадратичные отклонения значений по каждой строке файла раздельно. В скобках после каждого значения указать номер данного числа в строке исходного файла.

Делать предположение о длине наибольшей строки, о предельном количестве чисел в строке и предельном количестве строк в файле недопускается.

Задание 22.3. *Дан текстовый файл, в котором записаны слова, разделяемые пробелами и знаками препинания. Найти*

(a) начинающихся на заданную букву;

(b) заканчивающихся на заданную букву;

(c) начинающихся и заканчивающихся одной и той же буквой;

(d) содержащих заданную букву не менее одного раза;

(e) Число различных слов в файле.

(f) Число появлений и частоту встречаемости (процент от общего числа различных слов) каждого слова.

(g) Алфавитно-упорядоченный словарь всех слов в строке (без повторений).

(h) Словарь всех слов, упорядоченных по длине.

Делать предположение о длине наибольшей строки, о предельном количестве слов в строке и предельном количестве строк в файле недопускается.

Задание 22.4. *Даны два текстовых файла, содержащих словари — алфавитно-упорядоченный список слов (по одному слову в каждой строке). Требуется объединить эти два словаря с выводом результата в третий файл.*

(a) Исходные словари могут содержать повторения слов, количество повторений слова в результирующем словаре должно соответствовать суммарному количеству появлений данного слова в исходных словарях.

(b) Исходные словари не содержат повторений слов. Конечный словарь также не должен содержать повторений.

(c) Обобщить задачу на случай произвольного количества исходных словарей.

Временная сложность алгоритма должна быть $O(n)$, где n — суммарное количество строк в исходных файлах.

Задание 22.5. *Дан файл, содержащий информацию о результатах игр турнира между командами, в каждой строке записан результат одной игры в следующем формате: название первой команды (одно слово), пробел, количество очков, набранных первой командой, пробел, количество очков, набранных второй командой, пробел, количество очков, набранных второй командой.*

(a) Определить команду, набравшую наибольшее количество игровых очков.

(b) Определить команду, набравшую наибольшее количество турнирных очков (по регламенту турнира за победу в игре дается 3 турнирных очка, за ничью — 1 очко, за поражение очков не дается).

(c) Составить список команд, упорядоченный по убыванию количества набранных турнирных очков.

§8.2. Двоичные файлы

Глава 9. Структуры

§9.1. Модуль обработки матриц

Лабораторная работа №23

Обработка матриц

Задание 23.1. *Реализовать пользовательский тип данных "матрица" в виде структуры, представляя матрицу в виде*

(a) двумерного статического массива;

(b) двумерного динамического массива (одномерный массив ячеек и одномерный массив адресов строк);

(c) одномерного динамического массива.

Задание 23.2. *Реализовать функции модуля, обеспечивающие "существование" типа "матрица"*

(a) инициализация матрицы размером $n \times m$ без инициализации элементов;

(b) инициализация нулевой матрицы Z_{nm} размером $n \times m$;

(c) инициализация квадратной единичной матрицы E_n размера $n \times n$;

(d) удаление матрицы (освобождение памяти);

(e) быстрый доступ к указателю на элемент матрицы по паре индексов (константный и неконстантный);

(f) безопасное прочтение и установка элемента матрицы по паре индексов;

(g) создание копии матрицы в памяти (эквивалент операции присваивания).

Задание 23.3. *Реализовать следующие функции модуля для модификации матрицы:*

(a) умножение строки матрицы на число;

(b) перестановка строк матриц;

(c) сложение строк матрицы;

(d) выделение подматрицы.

Задание 23.4. *Реализовать следующие функции модуля для обработки матрицы:*

(a) умножения матрицы на число;

(b) вычисление суммы и произведения матриц;

(c) транспонирование квадратной матрицы;

(d) транспонирование прямоугольной матрицы;

(e) нахождение обратной матрицы квадратной матрицы;

(f) нахождение определителя матрицы.

Задание 23.5. *Реализовать функции модуля для нахождения нормы матрицы M размера $n \times m$:*

(a) $\|M\| = \max\limits_{i=0}^{n} \max\limits_{j=0}^{m} |M_{ij}|$,

(b) $\|M\| = \max\limits_{i=0}^{n} \sum\limits_{j=0}^{m} |M_{ij}|$,

(c) $\|M\| = \max\limits_{j=0}^{m} \sum\limits_{i=0}^{n} |M_{ij}|$,

(d) $\|M\| = \left(\sum\limits_{i=0}^{n} \sum\limits_{j=0}^{m} M_{ij}^2 \right)^{1/2}$.

Задание 23.6. *Используя модуль решить следующие задачи:*

(a) вычислить сумму матричного ряда для квадратной матрицы M размера $n \times n$:

$$\exp M = E_n + M + \frac{M^2}{2} + \frac{M^3}{6} + \cdots = \sum_{i=0}^{\infty} \frac{M^i}{i!},$$

заменив бесконечную сумму конечной, остановившись на таком слагаемом, матричная норма которого меньше некоторого напередзаданного ε от матричной нормы M (исключить прямое вычисление матричной степени и факториала, вычисляя очередное слагаемое через предыдущее);

(b) решить систему линейных уравнений (n уравнений, n неизвестных), снабдить тестирующую программу проверкой корректности решения с некоторой точностью ε.

§9.2. Эмуляция “базы данных”

Лабораторная работа №24
“База данных”

Задание 24.1. *Реализовать модуль обработки массива структур, описывающих некоторый объект реального мира не менее чем одним строковым, одним целочисленным и одним вещественным полем. Модуль должен включать следующие функции*

(a) добавление элемента;

(b) удаление элемента (по номеру);

(c) сортировка массива (по выбранному полю по возрастанию или убыванию);

(d) вывод элементов массива на экран и в текстовый файл;

(e) прочтение массива из текстового файла;

(f) вывод столбчатой диаграммы по выбранному полю с легендой (для разных элементов массива используется разный цвет столбцов, высоты столбцов пропорциональны значению поля, легенда содержит информацию о цвете столбца и значении текстового поля), высота диаграммы (т.е. высота наибольшего столбца) и ширина диаграммы не должны зависеть от значения поля массива, ширины столбцов для данной диаграммы должны быть одинаковы и определяться их количеством;

(g) вывод круговой диаграммы по выбранному полю с легендой (размер сектора пропорционален значению поля, весь круг соответствует сумме значений полей по всей элементам).

Глава 10. Динамические структуры данных

§10.1. Связные списки

Лабораторная работа №25
Стек и очередь

Задание 25.1. *Реализовать посредством статического массива*

(a) стек целых чисел;

(b) очередь целых чисел;

(c) двустороннюю очередь целых чисел.

Обеспечить сложность операций вставки и удаления в $\Theta(1)$.

Задание 25.2. *Реализовать посредством линейного связного списка*

(a) стек вещественных чисел;

(b) очередь вещественных чисел;

(c) двустороннюю очередь вещественных чисел.

Реализовать функции создания копии структуры в памяти.

Задание 25.3. *Дан односвязный список целых чисел, организованный посредством стека (очереди). Реализовать*

(a) поиск указателя на элемент с указанным значением (при наличии нескольких — первого в порядке добавления элементов), если такого элемента в списке нет, вернуть `NULL`

(b) поиск указателя на элемент с указанным индексом i, отсчитываемым в порядке добавления элементов, если такого элемента в списке нет, вернуть `NULL`

(c) вставку нового элемента на место, с указанным индексом i, отсчитываемым в порядке добавления элементов, если в списке менее $i-1$ элементов дополнить его нулями;

(d) вставку нового элемента, после элемента с указанным значением (при наличии нескольких — последнего такого в порядке добавления), если такой элемент не найден — заменить стандартной операцией помещения элемента;

(e) вставку после каждого элемента, содержащего квадрат его значения;

(f) удаление всех четных (по номеру добавления) элементов;

(g) удаление всех нечетных (по номеру добавления) элементов;

(h) удаление всех четных (по значению) элементов;

(i) удаление всех нечетных (по значению) элементов;

(j) удаление всех положительных (по значению) элементов;

(k) удаление всех отрицательных (по значению) элементов;

(l) удаление всех элементов, обладающих заданным свойством;

(m) “разворот” списка;

(n) разделение списка на 2: в один поместить нечетные (по значению), в другой — четные (по значению) элементы;

(o) разделение списка на 2: в один поместить положительные (по значению), в другой — отрицательные (по значению) элементы, нулевые элементы удалить;

Выделение дополнительной памяти (из кучи) недопускается.

Задание 25.4. *Реализовать калькулятор обратной польской записи посредством стека. На вход рабочей функции должна подаваться строка, на выход — результат или сообщение об ошибке (неверный формат числа, неверная функция или операция, недостаток операндов, недостаток операций или функций). Обрабатывать числа с плавающей точкой. Реализовать следующие возможности:*

(a) бинарные арифметические операции: сложение, умножение, вычитание, деление, возведение в степень;

(b) унарные операции: инверсию и взятие противоположного числа;

(c) тригонометрические функции (одного аргумента);

(d) гиперболические тригонометрические функции (одного аргумента);

(e) обратные тригонометрические функции (одного аргумента);

(f) обратные гиперболические тригонометрические функции (одного аргумента);

(g) функцию извлечения квадратного корня и экспоненты (одного аргумента);

(h) функции натурального, десятичного и двоичного логарифма (одного аргумента);

(i) функцию логарифма (двух аргументов);

(j) вычисление среднего арифметического двух чисел;

(k) вычисление среднего арифметического n чисел (n является ближайшим аргументом функции);

(l) вычисление среднеквадратичного отклонения n чисел (n является ближайшим аргументом функции).

Лабораторная работа №26
Циклический список

Задание 26.1. *Реализовать циклический односвязный список вещественных чисел. Обеспечить наличие возможностей*

(a) перехода к следующему элементу;

(b) вставка элемента после текущего;

(c) удаление элемента, следующего за текущим;

(d) создания копии списка;

(e) вывода всех элементов списка на экран.

Снабдить модуль пользовательским интерфейсом доступа к указанным возможностям посредством выбора пунктов меню.

Задание 26.2. *Реализовать циклический двусвязный список целых чисел. Обеспечить наличие возможностей*

(a) перехода к следующему и предыдущему элементу;

(b) вставка элемента после и перед текущим;

(c) удаление текущего элемента;

(d) создания копии списка;

(e) вывода всех элементов списка на экран.

Снабдить модуль пользовательским интерфейсом доступа к указанным возможностям посредством выбора пунктов меню.

Задание 26.3. *Реализовать модуль, обеспечивающий работу с циклическим двусвязным списком (задача 26.2 данных произвольного типа.*

§10.2. Деревья

Лабораторная работа №27
Двоичные деревья поиска

Задание 27.1. *Реализовать простое двоичное дерево поиска.*

Задание 27.2. *Реализовать красно-черное дерево.*

Задание 27.3. *Реализовать графическое отображение красно-черного дерева.*

Задание 27.4. *Реализовать консольное отображение красно-черного дерева.*

Задание 27.5. *Определить высоту дерева поиска.*

Задание 27.6. *Определить число элементов дерева поиска.*

Задание 27.7. *Определить число листьев дерева поиска.*

Задание 27.8. *Определить черную высоту красно-черного дерева.*

Задание 27.9. *Определить красную высоту красно-черного дерева.*

Задание 27.10. *Определить, является ли заданное двоичное дерево деревом поиска.*

Задание 27.11. *Определить, является ли заданное раскрашенное дерево красно-черном деревом.*