МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики

**А.А. Белеванцев, С.С. Гайсарян, Л.С. Корухова,
Е.А. Кузьменкова, В.С. Махнычев**

Семинары по курсу “Алгоритмы и алгоритмические языки”

(учебно-методическое пособие для студентов 1 курса)

Москва

2012

УДК 004.43+004.021
ББК 32.973-018я73
С30

Печатается по решению Редакционно-издательского совета факультета вычислительной математики и кибернетики Московского государственного университета имени М.В. Ломоносова

Рецензенты:
С.Ю. Соловьев, профессор
А.Н. Терехин, доцент

Белеванцев А.А., Гайсарян С.С., Корухова Л.С., Кузьменкова Е.А., Махнычев В.С. Семинары по курсу“Алгоритмы и алгоритмические языки” (учебно-методическое пособие для студентов 1 курса) – М.: Издательский отдел факультета ВМК МГУ имени М.В. Ломоносова (лицензия ИД N 05899 от 24.09.2001 г.); 2012. – 77 с.

ISBN 978-5-89407-497-9
ISBN 978-5-317-04299-8

В учебном пособии представлены материалы основной части семинарских занятий в поддержку курса лекций «Алгоритмы и алгоритмические языки» для студентов 1 курса факультета ВМК МГУ. Изучение такого курса предполагает усвоение студентами всех составляющих – лекции, семинарские занятия, практикум на компьютере, самостоятельная работа. В качестве базового языка в процессе обучения был использован язык Си99.

В пособии рассматриваются основные темы семинарских занятий. При рассмотрении тем кратко формулируются основные понятия и расставляются акценты, приводятся решения типовых задач, которые следует рассмотреть на семинарском занятии, а также даются задачи для самостоятельного решения.

Учебное пособие составлено с учетом опыта преподавания программирования для студентов факультета вычислительной математики и кибернетики МГУ и предназначено для студентов, изучающих основной курс программирования, а также для преподавателей и аспирантов.

Авторы выражают благодарность всем преподавателям практикума на 1 потоке 1 курса, которые на практике провели работу по реализации этого курса.

Издательский отдел
Факультета вычислительной математики и кибернетики
МГУ им. М.В. Ломоносова
Лицензия ИД N 05899 от 24.09.01 г.

119992, ГСП-2, Москва, Ленинские горы, МГУ имени М.В. Ломоносова, 2-й учебный корпус

Напечатано с готового оригинал-макета
в издательстве ООО "Макс Пресс"
Лицензия ИД N 00510 от 01.12.99
Подписано к печати 2012 г. Формат 60x90 1/16 Усл.печ.л. 1,5 Тираж nnn экз. Заказ

119992, ГСП-2, Москва, Ленинские горы, МГУ имени М.В. Ломоносова,
2-й учебный корпус, 627 к.
Тел. 939-3890, 939-3891. Тел./Факс 939-3891

ISBN 978-5-89407-497-9
ISBN 978-5-317-04299-8

Введение

В учебном пособии представлены материалы основной части семинарских занятий в поддержку лекционного курса «Алгоритмы и алгоритмические языки» для студентов 1 курса факультета ВМК МГУ. Курс состоит из трех разделов – элементы общей теории алгоритмов; язык программирования Си [1, 2], как пример алгоритмического языка; структуры данных и классические алгоритмы. В поддержку курса проводятся семинары (практикум) – два семинарских занятия в неделю, одно из которых посвящено разбору и усвоению нового материала, а второе – проверке домашних заданий в компьютерном классе. Предусмотрено проведение трех контрольных работ и коллоквиумов. Проверка домашних заданий и некоторых контрольных работ осуществляется с помощью системы автоматической проверки Ejudge (автор – Александр Чернов).

В пособии рассматриваются основные темы семинарских занятий. При рассмотрении тем кратко формулируются основные понятия и расставляются акценты. Приводятся решения типовых задач, которые следует рассмотреть на семинарском занятии. В качестве форм промежуточной отчетности по курсу предусмотрены коллоквиумы и зачет с оценкой по практикуму, итоговая отчетность – письменный экзамен по курсу. Типовые задачи коллоквиумов и экзаменов приведены в учебно-методическом пособии [3].

Наряду с "традиционным" домашним заданием, выполняемым в тетради, студентам даются задачи для самостоятельного решения с проверкой в системе автоматического тестирования. Задачи, прошедшие тестирование в системе, студент сдает преподавателю на занятии в компьютерном классе. Задачи домашних заданий опубликованы в отдельном пособии «Практические задачи по вводному курсу программирования» [4] . Темам 2 и 4 (операторы, выражения, целые типы) посвящены задачи первой домашней работы из пособия (раздел 1.1), теме 5 (функции) – задачи второй домашней работы (раздел 1.2), теме 6.1 (одномерные массивы) – третья домашняя работа (раздел 1.3). Тема 6.2 (строки) выделена в отдельную четвертую домашнюю работу (раздел 1.4). Темам 6.3 и 7 (многомерные массивы, указатели, динамическая память) посвящена пятая домашняя работа (раздел 1.5), а темам 3, 8, 9 – 10 (файлы, ввод-вывод, структурный тип и динамические структуры данных) – шестая домашняя работа (раздел 1.6).

Авторы выражают благодарность всем преподавателям практикума на 1 потоке 1 курса, которые на практике провели работу по реализации этого курса в 2010-2012 гг.

1. Язык программирования Си. Простейшие программы

Пример Си-программы. Структура программного файла. Функции и переменные. Объявления и определения функций, объявления переменных. Области видимости переменных: глобальные и локальные переменные. Директивы препроцессора.

Программа на языке Си состоит из объявления переменных или функций, а также определения функций (программа может быть записана в нескольких файлах, но для простоты мы будем рассматривать программы из одного файла). В примере программы на рис. 1 определяются функции `func` и `sum`. Также объявляется глобальная переменная `count` и несколько локальных переменных: `a` и `b` в функции `main` и `count` в функции `func`.

```
/* Простая программа на Си, показывающая определение
функций,  объявления  глобальных  и  локальных
переменных.
Это пример многострочного комментария */

// В Си-99 возможны такие однострочные комментарии

// Включение заголовочного файла
#include <stdio.h>

int count;  /* глобальная переменная */

void sum (int x, int y) /* определение функции */
{
  printf ("сумма %d и %d равна %d\n", x, y, x + y);
  count++;
}

void func (void)
{
  int count; /* локальная переменная */

  count = count - 2;
}

int main (void)
{
  int a, b;   // другие локальные переменные

  count = 0;
  scanf ("%d", &a); /* & как обозначение адреса */
  scanf ("%d", &b);

  sum (a, b);
  b = b * 3;
  sum (a, b);
```

```
    func ();
    printf ("Вызвали функцию sum %d раз\n", count);

    return 0;
}
```

Рис.1. Пример простой Си-программы

Глобальные переменные объявляются вне определения функций, а локальные – внутри функции или блока операторов. *Областью видимости* переменной является для локальной переменной – блок (составной оператор, см. раздел 3) или функция, в которой она объявлена, для глобальной переменной – программный файл, начиная со строки объявления. Внутри одной области видимости может быть лишь одна переменная с определенным именем, при этом возможно *перекрытие имен* – если имена локальной и глобальной переменной совпадают, то в локальной области видимости используется локальная переменная (на рис. 1 локальная переменная `count` перекрывает глобальную внутри функции `func`).

Выполнение программы начинается с функции `main`, которая имеет целочисленное возвращаемое значение. Это значение для функции `main` является кодом возврата программы для операционной системы, и есть договоренность, что завершившаяся успешно программа возвращает 0, а возврат ненулевого значения делается в случае ошибки. В теле функции выполняются вычисления и могут вызываться другие функции (на рис. 1 из функции `main` вызываются функции `sum` и `func`). В месте вызова функции должно быть известно ее объявление. Для вызова функций, определения которых содержатся в других программных файлах, их объявления включаются в текущий файл из заголовочных файлов с помощью директивы препроцессора `#include`. Так, на рис. 1 объявления функций ввода-вывода `scanf` и `printf`, определения которых не содержатся в текущем программном файле, включаются из файла `stdio.h` стандартной библиотеки Си.

При написании программ на практических занятиях и сдаче домашних занятий рекомендуется придерживаться одного стиля кодирования, например такого, которым записан пример на рисунке 1. Используемый стиль кодирования должен устанавливать следующие ограничения (в скобках даются некоторые примеры ограничений для программы на рисунке 1):

- размер отступов и максимальная длина строки (2 и 80 символов);
- имена переменных, типов и функций (осмысленные англоязычные имена, отражающие назначение переменных; для короткоживущих переменных, счетчиков цикла и т.п. допустимы общепринятые короткие имена и сокращения);
- объявление и инициализация переменных (переменные разных типов – на разных строках, инициализация переменных на строке объявления);
- стиль отступов и пробелов (блочные фигурные скобки на отдельных строках, пробелы между именем функции и открывающей скобкой списка параметров, пробелы между операндами бинарных операций).

2. Типы данных. Вычисление выражений

Типы данных языка Си. Целые знаковые и беззнаковые типы. Литералы, инициализация переменных. Присваивание переменных. Вычисление целочисленных выражений, приведение типов. Побочный эффект. Логические выражения. Точки последовательных вычислений. Побитовые операции. Условная операция. Старшинство операций.

Базовыми **типами** языка Си являются:
`char` (символьный),
`int` (целый),
`float` (с плавающей точкой),
`double` (двойной точности),
`void` (без значения).

Целочисленными типами являются типы `int` и `char`. К этим типам применимы модификаторы знаковости `signed/unsigned`. Если явно модификатор знаковости не указан, то целый тип `int` по умолчанию знаковый - `signed int`. К типу `int` применимы также модификаторы размера `short, long, long long`. По неубыванию размера целочисленные типы можно отсортировать как `char, short, int, long, long long`, при этом размер типа `int` не может быть менее 2 байт, типа `long` – не менее 4 байт, типа `long long` – не менее 8 байт. Для современной 64-битной архитектуры x86-64 размер типа `int` составляет 4 байта, `long` и `long long` – 8 байт.

Целочисленные константы записываются в виде чисел в десятичной, восьмеричной или шестнадцатеричной системе; основание системы счисления определяется префиксом, предшествующим цифрам числа: десятичные числа – без префикса (`100`), восьмеричные – с префиксом `0` (`077=63`), шестнадцатеричные – с префиксом `0x` или `0X` (`0x1F=31`). Тип целочисленной константы определяется буквенным суффиксом, приписываемым к цифрам числа: константы без суффикса имеют знаковый тип `int`, суффикс `L` соответствует типу `long`, суффикс `LL` – `long long`, а буква `U` (или `u`) – типу `unsigned`. Например, константы `1000u`, `34L`, `945LLU` имеют типы `unsigned int`, `long`, `unsigned long long` соответственно. При этом если константа без суффикса слишком велика, чтобы быть представлена типом `int`, она автоматически получит тип `long` или `long long`.

Символьные константы записываются в одинарных кавычках, например, `'a'`, `'5'`. Специальные символы, такие, как перевод строки, табуляция, а также сам символ одинарной кавычки могут задаваться с помощью символа `\` (обратный слэш), который требует специально толковать следующий за ним символ или последовательность символов (т.н. escape-последовательность). Так, перевод строки соответствует символу `'\n'`, табуляция – символу `'\t'`, одинарная кавычка – символу `'\''`. Можно указать код символа через последовательность шестнадцатеричных цифр `\x`, например, `'\x20'` в кодировке ASCII соответствует символу пробела. Символ с кодом 0 записывается как `'\0'`. Для поддержки символов в расширенных кодировках (Unicode) используется префикс L: `L't'` – символ `t` в Unicode.

Значения целочисленных типов хранятся в позиционной двоичной системе счисления. Знаковый и беззнаковый варианты одного и того же целого типа имеют один и тот же размер. При этом для беззнаковых `n`-битных типов возможные значения переменной данного типа лежат между `0` и $2^n - 1$. Для знаковых `n`-битных типов один бит отводится на знак, а остальные биты хранят модуль числа. Отрицательные числа, как

правило, представляются в дополнительном коде: если x<0, то n-1 бит, хранящие число x, представляются как $2^{n-1} - |x|$, при этом необходимо, чтобы $1 \le |x| \le 2^{n-1}$. Представимые в знаковом n-битном типе числа находятся между -2^{n-1} до $2^{n-1}-1$.

Задача.

Представить в дополнительном коде для 8-битного целого типа числа

0, 1, 12, 64, 127, -1, -4, -17, -128,

используя как двоичную, так и шестнадцатеричную системы счисления.

(*Ответы: 0 = 00000000 = 0x00, 1 = 00000001 = 0x01, 12 = 0x00001100=0x0C, 64 = 01000000 = 0x40, 127 = 01111111 = 0x7F, -1 = 11111111 = 0xFF, -4 = 11111100 = 0xFC, -17 = 1110 1111 = 0xEF, -128 = 10000000 = 0x80)*.

Объявление переменной состоит из описания типа переменной, ее имени и необязательной инициализации: `int x; unsigned long y=5; signed short z = - 2`. В качестве выражения для инициализации глобальных переменных может выступать константное выражение, т.е. такое выражение, которое может быть вычислено на этапе компиляции; для локальных переменных можно использовать любое выражение. При отсутствии инициализации глобальных переменных они обнуляются компилятором (*неявная инициализация*). Напротив, локальным переменным всегда необходимо присваивать значение вручную: неинициализированные локальные переменные имеют неопределенное значение при выполнении программы, а обращение к таким переменным, скорее всего, приведет к неверной работе программы и является частым источником ошибок.

При объявлении переменных часто используют спецификаторы классов памяти `static`, `extern` и квалификатор `const`:

- `const` – значение переменной не будет изменяться после инициализации, например, `const int size = 100`; При попытке изменить переменную произойдет ошибка на этапе компиляции.
- `static` – место под переменную будет выделено в статической памяти, доступ к переменной возможен во время всего выполнения программы, а ее инициализация выполняется до начала работы программы.
- `extern` – место под глобальную переменную выделяется при ее объявлении в другом файле, инициализация также выполняется в другом файле, доступ к переменной возможен во время всего выполнения программы.

Наиболее часто используемым выражением является операция присваивания "=". Левая часть присваивания должна обозначать объект памяти (будем рассматривать пока только имя переменной), правая – являться выражением (например, `a = b+c`, где `a, b, c` – целочисленные переменные). Результатом присваивания является изменение объекта памяти, т.е. *побочный эффект*. Обратите внимание, что присваивание является операцией и генерирует значение – результатом присваивания является значение левой части. Следовательно, возможно выписать цепочку присваиваний (`a = b = c+d`), при этом операция присваивания ассоциируется справа налево (т.е. сначала вычисляется выражение `c+d`, его результат записывается в переменную `b`, а потом – в переменную `a`, побочным эффектом этого выражения является изменение переменных `a` и `b`).

2.1. Вычисление выражений

Арифметические операции над целочисленными значениями (без побочного эффекта) бывают одноместные (одноместный или унарный минус "-" и плюс "+") и

двухместные (сложение "+", вычитание "−", умножение "*", деление нацело "/", остаток от деления нацело "%"). Обратите внимание, что запись "-23" считается константным выражением, а не целочисленной константой: в этом выражении операция унарного минуса применена к целочисленной константе 23.

При делении нацело результат всегда округляется в сторону нуля и выполняется равенство (a/b)*b + a%b = a, поэтому знак остатка совпадает со знаком делимого: -27/5 = -5, -27%5 = -2, -27%-5 = -2, 27%-5 = 2. Примеры выражений также содержатся в тексте программы на рис. 1.

2.1.1. О приведении типов операндов.

При вычислении арифметических операций с двумя операндами, а также при выполнении операции присваивания может автоматически выполняться неявное преобразование операндов (*приведение их к другому типу*). Для двуместных арифметических операций приведение типов выполняется по следующим правилам:

1) Если операнды имеют различные типы, то осуществляется их приведение к общему типу. Если один из типов операндов – вещественный, то второй операнд тоже приводится к вещественному типу, и общим типом будет наибольший вещественный тип из типов операндов (т.е. для float и double – double, для double и long double – long double и т.п.). Например, для получения вещественного результата операции деления 3/2 необходимо один из операндов сделать вещественным, например, добавив точку в одну из констант: "3./2". Иначе результатом операции будет целое число 1, а не вещественное число 1,5.

2). Если есть операнд целого типа короче, чем int (т.е. знаковый или беззнаковый вариант short или char), и все значения этого типа могут быть представлены как int, то он преобразуется к int; иначе - к unsigned int. Это преобразование называется «целочисленное расширение» (**integer promotion**) и выполняется для избежания потерь точности при вычислениях. Например, при сложении двух переменных типа short обе переменные приводятся к типу int, и само сложение выполняется в типе int.

3). Для операндов целых типов неявное приведение типов управляется целочисленным рангом приведения типа. Ранг выбирается таким образом, что ранг типа long long больше ранга long, который в свою очередь больше ранга int и т.д. до char. Ранг знакового и беззнакового варианта одного целого типа совпадают. При совпадении знаковости типов операндов операнд типа с меньшим рангом преобразуется к операнду типа с большим рангом. При несовпадении знаковости выполняется операция приведения, в которой также, как правило, операнд типа меньшего ранга преобразуется к типу операнда с большим рангом.

Арифметическая операция выполняется после неявного приведения типов, и типом результата является тип, к которому были приведены оба операнда.

Необходимо помнить, что при неявном приведении целых типов операнд знакового типа может быть приведен к операнду беззнакового типа и, если значение операнда знакового типа отрицательно, произойдет переполнение с возможной выдачей ошибочного результата вычисления.

Например, на современной системе с 4-байтовыми типами int и unsigned int пусть unsigned int u = 50; int i = -500; int res = i / u. Значение переменной res будет равно 85899335, т.к. при вычислении выражения i/u операнд i

будет приведен к беззнаковому типу и его значение будет равно $2^{32}-500$ вместо -500, а результат деления будет приведен обратно к знаковому типу int. Для избежания таких ошибок крайне рекомендуется выполнять арифметические операции над знаковыми целыми типами.

При присваивании переменных различных типов также выполняется неявное приведение типов: если не все значения типа правой части могут быть представлены типом левой части, то происходит отсечение старших битов или превращение знакового бита в значащий (для целых типов), либо округление или усечение числа для плавающих типов.

Например,

1) если `signed char c = -1`, `unsigned char uc` и `char` является 8-битным типом, то после `uc = c;` значение `uc` будет равно 255;

2) если `short s = 920` и тип `short` – 16-битный тип, то после `c = s;` значение `c` будет равно -104.

Явное приведение типов возможно с помощью выражения `(type) expr`: например, для получения вещественного результата деления в переменной `d` типа `double` суммы двух целых переменных `a` и `b` на два можно использовать выражение
`d = ((double) (a+b)) / 2.`

Часто возникает необходимость записать результат двухместной операции в один из операндов. В этом случае можно воспользоваться *укороченным присваиванием*:
`a = a op b` эквивалентно `a op= b`. Такие присваивания поддерживаются для всех двухместных операций.

Арифметическими операциями с побочным эффектом, помимо укороченных присваиваний, являются операции инкремента "++" и декремента "--" двух форм – префиксной (знак операции до операнда) и постфиксной (после операнда). Побочный эффект этих операций – увеличение (соответственно уменьшение) значения операнда на единицу, а результат операции – значение операнда до изменения (для постфиксной формы) либо после изменения (для префиксной формы). Например, если `int a = 5, b = 3`, то значение выражения `a-- + b` равно 8, побочный эффект – `a = 4`; значение выражения `a + --b` равно 7, побочный эффект – `b = 2`.

Операции сравнения (равно "==", не равно "`!=`") и отношения (больше ">", меньше "<", больше или равно ">=", меньше или равно "<=") генерируют результат типа `int`: единицу, если отношение истинно, и 0 в противном случае.

Двумя реже используемыми операциями являются операция последовательного вычисления "`,`" и условная операция "`?:`". В операции "`,`" операнды вычисляются слева направо, а результатом операции является значение последнего вычисленного выражения: в выражении `a = (b = 5, b + 2)` сначала переменной `b` присваивается значение 5, потом вычисляется выражение `b + 2`, равное 7, и результат заносится в переменную `a`. В операции "`expr1 ? expr2 : expr3`" сначала вычисляется выражение `expr1`, и если его значение отлично от нуля, то результатом всей операции является значение выражения `expr2`, иначе – значение выражения `expr3`. Условная операция, как и операция присваивания, ассоциируется справа налево, т.е. выражение

`a ? b : c ? d : e` эквивалентно `(a ? b : ( c ? d : e))`.

Точкой следования, или *точкой последовательных вычислений* (sequence point), называется момент во время выполнения программы (иногда говорят о месте в программе), в котором все побочные эффекты предыдущих вычислений закончены, а новых – не начаты. В корректной программе между двумя точками последовательных вычислений изменение значения переменной возможно не более одного раза, при этом старое значение читается только для определения нового. Такими точками в программе являются:

- конец полного выражения (т.е. для выражения `a + b - c*d` после вычисления всего выражения);
- при выполнении операции `x , y` – между вычислением `x` и `y`;
- при выполнении операции `z ? x : y` – между вычислением `z` и вычислением `x` либо `y`;
- при вызове функции – перед выполнением ее тела и после вычисления ее аргументов;
- при выполнении операций `x && y` и `x || y` – между вычислением `x` и вычислением `y`.

Из-за нарушения семантики точек последовательных вычислений выражения типа `(a=2) + (a=3)`, `i++ + ++i` являются некорректными (дважды модифицируется переменная), аналогично и выражения типа `a = b++ + b` (значение `b` читается как для определения нового значения `b`, так и для вычисления суммы). Значение таких выражений не определено (`undefined`), т.е. стандарт языка не требует от компилятора Си получить какое-то конкретное значение. Поэтому для разных компиляторов из-за особенностей выполняемых ими оптимизаций значения таких выражений будут различны. Использование неопределенных выражений крайне не рекомендуется: программа с ними будет получать неожиданные для программиста результаты или разные результаты в зависимости от случайных факторов.

2.2. Логические выражения

Логическими операциями являются отрицание "`!`", конъюнкция "`&&`" и дизъюнкция "`||`". Результатом этих операций является единица, если соответствующая логическая функция истинна, и ноль, если ложна. Для скалярных операндов (т.е. целочисленных, вещественных и указателей) истинным считается любое ненулевое значение, а ложью – лишь ноль.

Благодаря тому, что после вычисления первого операнда операций "`&&`" и "`||`" находится точка последовательных вычислений, становится возможным использования *«короткой»* логики: операнды вычисляются слева направо, и если результат операции уже известен (значение первого операнда равно 0 для конъюнкции и 1 для дизъюнкции), вычисление второго операнда не производится.

Например, в выражении `if (n != 0 && b > a/n)` деления на ноль не возникнет, так как в этом случае вычисления второго операнда конъюнкции не будет. В другом примере `if ((c = getchar ()) != EOF && isprint (c))` вызов функции `isprint` произойдет только в том случае, если функция `getchar` вернет значение, отличное от `EOF`, при этом гарантируется, что переменная `c` будет уже изменена в ходе вычисления побочных эффектов первого операнда конъюнкции.

Задача. Написать эквивалентное выражение, не содержащее операции отрицания "!":

```
а) ! (a>b)
б) ! (2*a == b+4)
в) ! (a<2 || a>5)
г) ! (a<b && c<d)
д) ! (a<1 || b<2 && c<3).
```

Решение.

а) `a <= b`
б) `2*a != b+4`
в) `a>=2 && a<=5`
г) `a>=b || c>=d`
д) `a>=1 && (b>=2 || c>=3).`

2.3. Побитовые операции

Побитовые, или поразрядные, операции работают над двоичным представлением числа. Одноместными побитовыми операциями является инверсия или побитовое отрицание "~", двухместными – побитовое И "&", побитовое включающее ИЛИ "|", поразрядное исключающее ИЛИ "^", побитовые сдвиги влево "<<" и вправо ">>". Сдвиг на отрицательное число бит или на число, превосходящее ширину первого операнда, не определен.

Нужно отметить, что при сдвиге вправо беззнакового операнда освобождающиеся биты заполняются нулями, тогда как для знаковых операндов возможно заполнение как нулями (логический сдвиг), так и значением знакового бита (арифметический сдвиг) – выбор точного поведения зависит от реализации. Поэтому для побитовых вычислений рекомендуется всегда использовать беззнаковые типы.

Пример. Пусть переменная `x` имеет тип `unsigned int`. Чтобы обнулить все биты `x`, кроме трех младших, необходимо использовать побитовую операцию `&`, взяв в качестве второго параметра число, у которого лишь три младших бита установлены: это число 7, т.е. ответом является выражение `x & 7`. Аналогично, для обнуления всех бит, кроме трех младших, вторым параметром побитового И должно быть число с установленными всеми битами, кроме трех младших, то есть `~7`, все выражение есть `x & ~7`. Для установки двух младших бит используется включающее ИЛИ: `x | 3`. Необходимую константу можно также получить сдвигом: для установки пяти младших бит вместо запоминания константы 31 используют выражение `x | ((1 << 5) - 1)`.

2.4. Старшинство операций

Старшинство операций задается таблицей приоритетов (см. рис. 2) и, как правило, соответствует ожиданиям программиста, избавляя его от необходимости писать лишние скобки в сложных выражениях.

Операции	Ассоциативность
( ) [ ] -> .	Слева направо
! ~ ++ -- + - sizeof (type) * &	Справа налево
*** / %**	Слева направо
+ -	Слева направо
<< >>	Слева направо
< <= > >=	Слева направо
== !=	Слева направо
&	Слева направо
^	Слева направо
|	Слева направо
&&	Слева направо
||	Слева направо
?:	Справа налево

= += -= *= /= %= &= ^= |= <<= >>=	Справа налево
,	Слева направо

Рис.2. Старшинство операций в Си.

Приоритет операций именования памяти (доступа к членам структуры, элементам массива, вызова функции) наивысший. Далее, приоритет одноместных операций выше, чем двухместных. Приоритет двухместных мультипликативных операций, в том числе логических и побитовых, выше, чем соответствующих аддитивных: приоритет "`* / %`" выше, чем у "`+ -`"; приоритет "`&&`" выше "`||`" , а приоритет "`&`" выше, чем "`^`" и "`|`". Приоритет отношений ("`< <= > >= == !=`") ниже, чем арифметических операций и сдвигов. Приоритет операции присваивания, в том числе укороченных присваиваний, самый низкий, за исключением операции “запятая”.

Пример. В выражении

`x = 3*a + b/c >= d` скобки естественно расставляются как

`x = ((3*a) + (b/c)) >= d`. Тем не менее, рекомендуется употреблять скобки, если они улучшают ясность и читаемость кода, либо в случае сомнений программиста.

2.5. Задачи для самостоятельного решения

2.5.1. Целой переменной `k` присвоить значение, равное третьей от конца цифре в записи целого положительного числа `x`.

2.5.2. Целой переменной `k` присвоить значение, равное сумме цифр в записи целого положительного трехзначного числа `x`.

2.5.3. Целой переменной `k` присвоить значение, равное первой цифре дробной части в записи вещественного положительного числа `x`.

2.5.4. Определить число, полученное выписыванием в обратном порядке цифр заданного целого трехзначного числа.

2.5.5. Используя условную операцию “?:”, написать выражение для вычисления:

а) модуля целого числа x,

б) максимума двух целых чисел,

в) максимума трех целых чисел.

2.5.6. Пусть `int a = 5, int b = 7`. Чему будет равно значение выражения и побочные эффекты?

а) `(a += 5) * (b -= 3)`

б) `--b / (a++ - 3)`

в) `(a -= 2) || 47 / (b - 7)`

г) `(a *= b) + (b *= a)`

2.5.7. Пусть `int x = 10 ; int y = 20;`

Для приведенного выражения указать его значение и побочные эффекты (если они есть) либо “ошибка”, если выражение ошибочно

а) `x += y , y += x, y, x`

б) `y += (x = 1) + (x = 2)`

в) `x +=((y=1)&&(y=2))`

г) `y %= x / 6`

д) `x || ++y`

е) `x ? !x : y`

2.5.8. Пусть определены переменные `k` и `x`, имеющие тип `unsigned int`. Считая, что в `x` находится трехзначное число, выпишите присваивание, которое поместит в `k` это число в «перевернутом» виде. Например, если в `x` содержалось 123, то в `k` следует поместить число 321.

2.5.9. Пусть определены переменные `double d; int i;` Укажите какие значения будут иметь переменные `i` и `d` после вычисления

```
i = d = 10/4;
```

Пояснение. В задачах 2.5.10-2.5.15 считать, что тип `unsigned int` занимает 32 бита, а тип `char` – 8 бит.

2.5.10. «Упаковать» четыре символа в 32-битное беззнаковое целое.

2.5.11. «Распаковать» 32-битное беззнаковое целое число в четыре символа.

2.5.12. Реализовать операции для типа «множество» (отображение номеров элементов множества на разряды слова, количество элементов множества задано размером слова): а) объединение, пересечение, вычитание, дополнение, сравнение множеств; б) добавить/удалить элемент в/из множества; является ли множество пустым; в) выбрать произвольный элемент из множества.

2.5.13. Не используя дополнительных переменных, поменять местами значения двух переменных типа int.

2.5.14. Поменять знак переменной `x` типа `int`.

2.5.15. Инвертировать 5 младших битов переменной `x` типа `unsigned int`, остальные биты оставить без изменения.

3. Ввод/вывод.

Символьный и форматный ввод-вывод данных. Операторы: выражение-оператор, условный оператор, оператор выбора, операторы цикла, оператор перехода, составной оператор.

Функции ввода-вывода в Си являются частью стандартной библиотеки языка, для их использования нужно подключать заголовочный файл `stdio.h` (см. раздел 1). В этом разделе познакомимся с некоторыми функциями, оперирующими со стандартными потоками ввода и вывода (которые обычно связаны с клавиатурой и дисплеем соответственно). Функции стандартной библиотеки, предназначенные для работы с файлами, описаны в разделе 9.

Функциями символьного ввода-вывода (т.е. считывающими и записывающими один символ) являются функции `getchar` и `putchar` со следующими объявлениями: `int getchar(void);` и `int putchar(int c);`. Обе функции возвращают считанный либо записанный символ типа `unsigned char`, приведенный к `int`, либо специальное значение `EOF`, если достигнут конец потока ввода либо произошла другая

ошибка ввода-вывода. Обратите внимание, что из-за необходимости различать верно считанный символ и состояние ошибки функции не могут возвращать значение типа `char` или `unsigned char`, поэтому используется значение типа `int`. Типичный цикл, считывающий по одному символу из стандартного потока ввода до достижения его конца, выглядит так:

```
int c;
while ((c = getchar ()) != EOF)
{
  /* обработать следующее значение c */
  ...
}
```

Функции форматного ввода-вывода `scanf` и `printf` (с прототипами `int scanf(const char *format, ...);` и `int printf(const char *format, ...);`) могут считывать и записывать данные любых базовых типов и строки согласно заданной *форматной строке*, которая является первым аргументом функций. Форматная строка задает количество и тип значений, которые необходимо считать либо записать, а остальные аргументы являются указателями на объекты памяти, куда нужно поместить считанные значения, либо выражениями, вычисляющие значения для записи. Формат одного значения задает спецификатор ввода-вывода, начинающийся с символа `%`. Часто используемыми спецификаторами являются:

- `%d`, `%ld`, `%lld` – напечатать/считать число типа `int`, `long`, `long long`;
- `%u`, `%lu`, `%llu` – напечатать/считать число типа `unsigned`, `unsigned long`, `unsigned long long`;
- `%f`, `%Lf` – напечатать число типа `double`, `long double`;
- `%f`, `%lf`, `%Lf` – считать число типа `float`, `double`, `long double`;
- `%c` – напечатать/считать символ;
- `%s` – напечатать/считать строку (при вводе строка считывается до первого пробельного символа);
- `%%` – напечатать знак процента.

Остальные символы форматной строки при выводе сохраняются в том же виде, что может использоваться для выдачи сообщений или перехода на новую строку. Например, следующий вызов выводит два целых числа и их сумму, переводя вывод на новую строку: `printf ("%d %d %d\n", a, b, a + b);`. Усложнением спецификаторов можно более точно настроить вид выводимых чисел, например, спецификатор `%.5f` выводит число типа `double` с пятью знаками после запятой.

Функция `scanf` возвращает количество правильно прочитанных (согласно форматной строке) аргументов, что широко используется для проверки корректности ввода. Следующий участок кода считывает три значения типа `double` и печатает сообщение, если при вводе произошла ошибка:

```
if (scanf ("%lf%lf%lf", &x, &y, &z) != 3)
printf ("Ошибка ввода: нужно ввести три вещественных числа\n");
```

Необходимо обращать особое внимание на передачу аргументов в функцию `scanf` для записи значений по указателю, а также точному соответствию типа указателя и спецификатора ввода, так как в случае ошибки считанные функцией значения будут записаны в неверные области памяти, что приведет к ошибочной работе программы или ее краху. Для контроля использования функций форматного ввода-вывода целесообразно при компиляции с использованием GCC включать предупреждения семейства `-Wformat`.

3.1. Задачи для самостоятельного решения

3.1.1.* Пусть определены переменные :

```
double d; float f; long lng; int i; short s;
```

Что будет напечатано в результате выполнения следующего фрагмента программы?

```
...
```

а)
```
s = i = lng = f = d = 100/3;
printf("s = %hd i = %d lng = %ld f = %f d = %f\n", s, i, lng, f, d);
```

б)
```
d = f = lng = i = s =100/3;
printf("s = %hd i = %d lng = %ld f = %f d = %f\n", s, i, lng, f, d);
```

в)
```
s = i = lng = f = d = 1000000/3;
printf("s = %hd i = %d lng = %ld f = %f d = %f\n", s, i, lng, f, d);
```

г)
```
d = f = lng = i = s =1000000/3;
printf("s = %hd i = %d lng = %ld f = %f d = %f\n", s, i, lng, f, d);
```

д)
```
lng = s = f = i = d =100/3;
printf("s = %hd i = %d lng = %ld f = %f d = %f\n", s, i, lng, f, d);
```

е)
```
f = s = d = lng = i = (double)100/3;
printf("s = %hd i = %d lng = %ld f = %f d = %f\n", s, i, lng, f, d);
```

ж)
```
s = i = lng = f = d = 100/(double)3;
printf("s = %hd i = %d lng = %ld f = %f d = %f\n", s, i, lng, f, d);
```

з)
```
f = s = d = lng = i = (double)100/3;
printf("s = %hd i = %d lng = %ld f = %f d = %f\n", s, i, lng, f, d);
```

и)
```
i = s = lng = d = f = (double)(100/3);
printf("s = %hd i = %d lng = %ld f = %f d = %f\n", s, i, lng, f, d);
```

3.1.2. Пусть определены переменные :

```
double d = 3.2, x; int i = 2, y;
```

Что будет напечатано в результате выполнения следующего фрагмента программы?

а)
```
x = ( y = d / i ) * 2;
  printf ("x = %f ;y = %d\n", x, y);
```

б)
```
x = ( y = d / i ) * 2;
  printf ("x = %d ;y = %f\n", x, y);
```

в)
```
y = ( x = d / i ) * 2;
  printf ("x = %f ;y = %d\n", x, y);
```

г)
```
y = d * ( x = 2.5 / d);
  printf ("x = %f; y = %d\n", x, y);
```

д)
```
x = d * ( y = ( (int)2.9 + 1.1) / d;
```

*Задачи из задачника [5]

```
printf ("x = %d y = %f\n", x, y);
```

4. Операторы

Для организации вычислений в языке Си используются операторы. Простейшим оператором является выражение-оператор вида "expression;": например, `a = b;` `a++;` или даже `5+1;` – это корректные операторы. Составной оператор `{}` позволяет группировать несколько операторов вместе. Условный оператор имеет вид **`if`** `(expr) stmt;` **`else`** `stmt;`, причем нужно отметить, что ветка **`else`** всегда относится к ближайшему **`if`**: два приведенных участка кода эквивалентны.

```
if (x > 2)
  if (y > z)
    y = z;
  else
    z = y;
```

```
if (x > 2) {
  if (y > z) {
    y = z;
  } else {
    z = y;
  }
}
```

Оператор выбора **`switch`** записывается как

```
switch (expr)
{
  case const-expr: stmt;
  case const-expr: stmt;
  default: stmt;
}
```

и позволяет выполнить один из операторов в зависимости от значения целочисленного выражения либо выполнить некоторый оператор по умолчанию (метка `default`), если ни один из вариантов значения не предусмотрен. Можно перечислять несколько вариантов значений для одного оператора. После того, как одно из значений подошло, происходит переход по соответствующей метке `case` и начинается выполнение тела `switch` с оператора после этой метки. Остальные операторы `switch` также выполняются в порядке их записи, поэтому для прекращения выполнения оператора выбора обязательно нужно использовать оператор `break`:

```
switch (tag)
{
  case 2: a *= b; /* Далее выполнится a += b; из следующей
ветки */
  case 3: a += b; break; /* А здесь произойдет выход */
  case 4: a -= b; break;
  default: break;
}
```

Операторами цикла в Си являются `while`, `for`, и `do .. while`. Цикл `while` имеет вид `while (expression) stmt;` и выполняет оператор `stmt`, пока вычисление управляющего выражения `expression` дает ненулевое значение, при этом первая проверка выражения производится до первого выполнения оператора. Цикл `do .. while` записывается как `do { stmt; } while (expression);` и аналогичен циклу `while`, но проверка условия выхода из цикла делается после оператора `stmt`, т.е. тело цикла исполняется хотя бы один раз.

Цикл `for` является самым мощным видом цикла в Си и записывается как `for (expr1; expr2; expr3) stmt;`. Перед началом выполнения цикла

вычисляется выражение `expr1`, как правило, используемое для инициализации счетчиков цикла. При этом переменную-счетчик можно объявить прямо в заголовке цикла. Тело цикла – оператор `stmt` – выполняется до тех пор, пока значение выражения `expr2` оказывается ненулевым, это выражение вычисляется перед началом каждой итерации цикла. После окончания итерации цикла вычисляется выражение `expr3`, обычно содержащее обновление счетчика цикла. Примером цикла `for` является вычисление n-й степени числа `x: for (int i = 1, pow = 1; i <= n; i++) pow *= x;`. Цикл `for` часто используется программистами из-за выразительности и наглядности: все управляющие циклом конструкции сосредоточены в его заголовке, причем нет ограничений на их содержание.

Все три выражения из заголовка цикла `for` могут быть опущены, при этом значение выражения `expr2` всегда считается истинным. Например, вечный цикл на языке Си может быть записан как `for ( ; ; ) { ... }`.

Для досрочного выхода из цикла, как и в случае оператора выбора, может быть использован оператор `break`. Этот оператор прерывает лишь самый внутренний цикл из гнезда циклов. Оператор `continue` используется для немедленного перехода к следующей итерации цикла, пропуская оставшуюся часть тела цикла, при этом выражение `expr3` не пропускается и выполняется до начала следующей итерации. Таким образом, при использовании `continue` нет необходимости дополнительно к коду заголовка цикла обновлять счетчик цикла.

Задача. С помощью цикла `while` напишите цикл, эквивалентный описанному общему виду цикла `for`. Что, если тело цикла содержит оператор `continue`?

Оператор перехода по метке `goto label;` используется для организации сложного потока управления. В качестве метки может выступать идентификатор, областью видимости которого оказывается вся функция. Место перехода помечается в программе как `label:`. Нужно избегать неоправданного применения оператора `goto`.

4.1. Задачи для самостоятельного решения

4.1.1. Подсчитать количество натуральных чисел `n` (`111 <= n <= 999`), в записи которых есть две одинаковые цифры.

4.1.2.. Подсчитать количество натуральных чисел `n` (`102 <= n <= 987`), в которых все три цифры различны.

4.1.3. Подсчитать количество натуральных чисел `n` (`11 <= n <= 999`), являющихся палиндромами, и распечатать их.

4.1.4. Подсчитать количество цифр в десятичной записи целого неотрицательного числа `n`.

4.1.5. Определить, верно ли, что куб суммы цифр натурального числа `n` равен n^2.

4.1.6. Определить, является ли натуральное число n степенью числа 3.

4.1.7. Дано натуральное число `n`. Получить все его натуральные делители.

4.1.8. Дано натуральное число `n`. Получить наименьшее число вида 2^r, превосходящее `n`.

4.1.9. Распечатать первые `n` простых чисел.

4.1.10. Распечатать первые `n` чисел Фибоначчи

($f_0 = 1;\ f_1 = 1;\ f_{k+1} = f_{k-1}+f_k;\ k = 1, 2, 3, \ldots$)

5. Функции

5.1. Функции

Понятие функции. Определение, объявление, вызов функции. Передача параметров в функцию. Понятие указателя. Задачи.

5.1.1. Понятие функции

В разделе 1 при описании простейших программ на языке Си уже были рассмотрены примеры использования функций. Рассмотрим это базовое для языка Си понятие более подробно. Под функцией в языках программирования подразумевается специальным образом оформленный фрагмент программы, описывающий решение некоторой подзадачи, как правило, неоднократно выполняемой при работе программы. Использование функций при разработке программ позволяет получить более наглядный, компактный и легче понимаемый код программы, а также облегчает повторное использование кода за счет использования в программах уже ранее описанных функций. Тем самым устраняется дублирование работы и ускоряется процесс разработки программы.

Определение функции в языке Си имеет вид:

```
<тип> <имя> (<список формальных параметров>)
{
<тело функции>
}
```

где `<тип>` задает тип возвращаемого функцией значения, `<список формальных параметров>` содержит перечисление параметров функции с обязательным указанием типа для каждого отдельного параметра, `<тело функции>` содержит объявление локальных переменных и операторы, реализующие алгоритм данной функции. Если функция возвращает некоторое значение в качестве результата своей работы, то в теле функции обязательно должен присутствовать оператор:

```
return <выражение>;
```

который завершает выполнение функции и возвращает вычисленное значение выражения в вызывающую функцию. При вычислении выражения по мере необходимости выполняется приведение типа к объявленному типу результата.

В качестве примера приведем определение функции, вычисляющей максимум двух целых чисел:

```
int
max (int a, int b)
{
  return (a > b) ? a : b;
}
```

Если функция не возвращает никакого значения, то в качестве типа возвращаемого значения указывается `void`. Оператор `return` в таких функциях может отсутствовать (в этом случае выполнение функции завершается по достижению конца тела функции, т.е. закрывающей скобки `}`), либо присутствовать, но не содержать никакого выражения (в

этом случае выполнение функции завершается немедленно по достижению оператора `return`). Например:

```
void
max1 (int a, int b)
{
  printf ("%d\n", (a > b) ? a : b);
}
```

Если функция не имеет параметров, то в качестве списка ее параметров указывается `void`. Например:

```
int
max2 (void)
{
  int a, b;
  scanf ("%d %d", &a, &b); // ввод значений a и b
  return (a > b) ? a : b;
}
```

Определению функции в программе может предшествовать ее объявление, заданное в виде прототипа этой функции, где указывается имя функции, тип возвращаемого ею значения и формальные параметры функции. Например:

```
int
max (int a, int b); // прототип функции max
```

Имена формальных параметров в прототипе можно не указывать. Перед использованием функции она должна быть обязательно определена или объявлена в программе, т.е. в месте вызова функции должен быть известен, по крайней мере, ее прототип.

Вызов функции имеет вид:

```
<имя> (<список фактических параметров>)
```

Между списками фактических и формальных параметров функции должно быть соответствие по типу и порядку следования параметров в этих списках. Обратите внимание, что наличие скобок обязательно даже в случае, когда функция не имеет параметров. Параметры передаются только по *значению*, то есть в отдельные области памяти, соответствующие формальным параметрам, копируются значения переменных или выражений, являющихся фактическими параметрами, и функция не влияет на фактические параметры (в частности, не может изменять их значения). Для обеспечения возможности изменять значения передаваемых в качестве параметров объектов необходимо использовать указатели, при этом моделируется передача параметров по *ссылке*: формальные параметры связываются (ссылаются) с той же областью памяти, что и фактические.

5.1.2. Понятие указателя

Указатель – это переменная, значением которой является адрес некоторого объекта. Пример определения указателя:

```
int x, *p;
```

Здесь x - переменная целочисленного типа, p - указатель на объект целочисленного типа. Указатель p может содержать адрес любого объекта целочисленного типа.

Для указателей определены операции & (взятие адреса объекта) и * (операция разыменования, взятие значения по адресу). Например, при выполнении оператора:

```
p = &x;
```

в переменной p будет зафиксирован адрес переменной x, а при выполнении оператора:

```
*p = 10;
```

объекту, на который ссылается указатель p, будет присвоено значение 10.

Использование указателей в качестве параметров функции проиллюстрируем на примере следующей задачи.

Задача 1. Описать функцию, меняющую местами две целочисленные переменные.

Попробуем решить задачу без использования указателей, описав функцию swap:

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

Проверим полученный результат:

```
int x = 1, y = 2;
swap (x, y);
printf ("%d %d\n", x, y);
```

На стандартный поток вывода будет выведено:

```
1 2
```

Следовательно, значения переменных не изменились! Это и понятно, потому что параметры функции swap передаются по значению и, следовательно, поменялись местами только копии параметров x и y внутри функции, при этом никакого влияния на сами фактические параметры x и y не произошло.

Для достижения нужного эффекта необходимо в качестве параметров функции использовать указатели на соответствующие переменные, т.е. передавать в функцию адреса переменных, значения которых требуется поменять:

```
void
swap (int *pa, int *pb) // обмен местами *pa и *pb
{
  int tmp = *pa;
  *pa = *pb;
  *pb = tmp;
}
```

Убедимся в работоспособности данного варианта функции `swap`:

```
int x = 1, y = 2;
swap (&x, &y); // передаем в функцию адреса переменных x и y
printf ("%d %d\n", x, y);
```

На стандартный поток вывода будет выведено:

```
2 1
```

Следовательно, обмен значений переменных действительно произошел. Работу данного варианта функции `swap` иллюстрирует следующий рисунок:

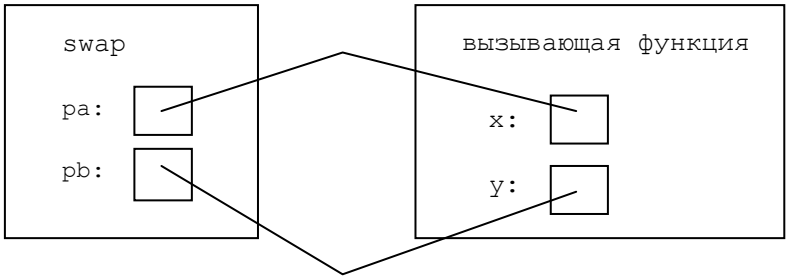

Рис. 3. Работа функции `swap`

Таким образом, для обеспечения непосредственного доступа к объектам в вызывающей функции (в частности, для модификации их значений) соответствующие параметры должны передаваться в виде указателей на эти объекты.

5.2. Рекурсивные функции

Понятие рекурсии. Примеры рекурсивных определений. Рекурсия и итерация. Методика разработки рекурсивных функций. Задачи.

5.2.1. Рекурсия

Рекурсия подразумевает использование в определении некоторого понятия самого определяемого понятия. В качестве примера рекурсивного определения рассмотрим определение понятия <выражение> :

<выражение> ::= <терм> | <выражение> <операция> <выражение>
<терм> ::= <число> | <переменная> | <обращение к функции> | (<выражение>)

Здесь в определении понятия <выражение> используется само определяемое понятие.

Рекурсивным может быть и определение функции. Например, функцию вычисления n! рекурсивно можно определить так:

$$n! = \begin{cases} 1, & n = 0, \\ n * (n - 1)!, & n>0 \end{cases}$$

В этом определении *n!* определяется через *(n - 1)!*, т.е. также имеется рекурсивное определение.

Заметим, что ту же функцию *n!* можно определить и без использования рекурсии как произведение последовательных целых чисел от *1* до *n*. Такому определению будет соответствовать итеративная реализация функции, использующая цикл для вычисления произведения указанных чисел.

Задача 1. Привести рекурсивную и итеративную реализацию функции *n!*.
Воспользуемся приведенными выше определениями функции *n!*.

Рекурсивная реализация:
```
unsigned int
fact (unsigned int n)
{
  return (n < 2) ? 1 : n * fact (n-1);
}
```
Промоделируйте вычисление `fact(3)`.

Итеративная реализация:
```
unsigned int
fact (unsigned int n)
{
  int i, f = 1;
  for (i = 2; i <= n; i++) {//цикл для вычисления факториала
    f *= i;
  }
  return f;
}
```
Сравните предложенные реализации с точки зрения их эффективности.

Задача 2. На стандартном потоке ввода задан текст (последовательность символов, заканчивающаяся точкой, точка в текст не входит). На стандартный поток вывода вывести этот текст в обратном порядке.

Приведем рекурсивную реализацию решения задачи.

```
void
print_rev (void)
{
  char ch;
  scanf ("%c", &ch); //ввод очередного символа
  if (ch != '.') {
    print_rev (); //рекурсивный вызов для обработки
                  //оставшихся символов
    printf ("%c", ch); //вывод символа
  }
}
```

В предложенной реализации для хранения очередного вводимого символа используется локальная переменная `ch`. Реализация механизма обработки вызовов функций предусматривает для каждого вызова функции `print_rev` создание своего экземпляра этой локальной переменной и размещение ее в стеке (в стековом фрейме функции – области памяти в стеке, используемой для работы функции). Аналогичным образом обрабатываются и параметры функции. При выходе из функции выделенная ей в стеке память освобождается.

Промоделируйте работу функции `print_rev` на примере текста `"abcd."`.

При разработке рекурсивных функций целесообразно следовать следующей методике. Как правило, тело рекурсивной функции содержит условный оператор `if`, одна из ветвей которого (нерекурсивная) обеспечивает выход из рекурсии (эта ветвь реализует обработку простейших случаев исходных данных), а вторая (рекурсивная) реализует обработку всех остальных случаев исходных данных, используя при этом рекурсивный вызов самой определяемой функции. В рассмотренных выше примерах выход из рекурсии обеспечивается при обработке ситуации `n < 2` для функции вычисления факториала и ситуации ввода точки для функции `print_rev`. По рекурсивной ветви (на примере функции `print_rev`) осуществляется непосредственная обработка одного из исходных значений (в данном случае текущего символа), для обработки оставшихся значений используется рекурсивный вызов функции.

Сравнивая рекурсивный и итеративный подходы к реализации функции, следует отметить, что несомненным достоинством рекурсивной реализации является ее наглядность и компактность. Особенно хорошо это видно при обработке структур данных, предполагающих внутри себя рекурсию, таких как списки, деревья и т.д. Однако за счет накладных расходов на обеспечение рекурсивных вызовов функции рекурсивная реализация проигрывает итеративной в эффективности как по памяти, так и по быстродействию.

5.3. Задачи для самостоятельного решения

Функции

5.3.1. Описать функцию, которая проверяет, обладает ли целое неотрицательное число `n` указанным свойством. В случае положительного ответа функция возвращает значение 1 и 0 в противном случае. Свойства:

a) `n` является полным квадратом,
b) `n` является простым числом,
c) `n` является степенью числа 3.

5.3.2. Программа. На стандартном потоке ввода задано число `n` (`n > 0`) и последовательность из `n` целых чисел. Описав соответствующую функцию, найти количество элементов последовательности, являющихся:

a) числами Фибоначчи (задаются соотношением
$f_0 = 1,\ f_1 = 1,\ f_{k+1} = f_{k-1} + f_k,\ k > 1$),
b) палиндромами.

5.3.3. Пусть n – целое неотрицательное число. Описать функцию от параметра n, которая находит:

a) сумму цифр этого числа,
b) максимальную цифру в десятичной записи этого числа,
c) количество нечетных цифр в десятичной записи этого числа.

Рекурсивные функции

5.3.4. Рекурсивно описать функцию вычисления x^n для вещественного x $(x \neq 0)$ и целого n:

$$x^n = \begin{cases} 1 & при \quad n = 0, \\ 1 / x^{|n|} & при \quad n < 0, \\ x * x^{n-1} & при \quad n > 0. \end{cases}$$

Протестировать эту функцию на подходящих наборах входных данных.

5.3.5. Рекурсивно описать функцию *c(m, n)*, где $0 \le m \le n$, для вычисления биномиального коэффициента C_n^m по формуле:

$$C_n^0 = C_n^n = 1, \quad C_n^m = C_{n-1}^m + C_{n-1}^{m-1} \quad \text{при} \quad 0 < m < n.$$

5.3.6. Программа. Рекурсивно описать функцию вычисления *НОД(n, m)* - наибольшего общего делителя неотрицательных целых чисел *n* и *m*, основанную на соотношении *НОД(n, m) = НОД(m, r)*, где *r* – остаток от деления *n* на *m*. С ее помощью найти наибольший общий делитель натуральных чисел *a* и *b*. Сравнить эффективность рекурсивной и нерекурсивной реализации функций вычисления *НОД*.

5.3.7. Программа. Рекурсивно описать функцию вычисления $НОД(n_1, n_2, n_3, ..., n_m)$, где $m > 2$, воспользовавшись для этого соотношением:

$$НОД(n_1, n_2, n_3, ..., n_k) = НОД(НОД(n_1, n_2, n_3, ..., n_{k-1}), n_k), \; k = 3, 4, ..., m.$$

С помощью этой функции найти $НОД\ (a_1, a_2, a_3, ..., a_{10})$.

5.3.8. Программа. Числа Фибоначчи задаются соотношением:

$$f_0 = 1, f_1 = 1, f_{n+1} = f_{n-1} + f_n, \; n > 1.$$

Рекурсивно описать функцию вычисления n-ого числа Фибоначчи и с ее помощью вычислить 10-е число Фибоначчи.

5.3.9. Пусть *n* – целое неотрицательное число. Рекурсивно описать функцию от параметра *n*, которая находит:

a) сумму цифр этого числа,
b) максимальную цифру в десятичной записи этого числа,
c) старшую (самую левую) цифру в десятичной записи этого числа.

5.3.10. На стандартном потоке ввода задан текст (последовательность символов, заканчивающаяся точкой, точка в текст не входит). Рекурсивно описать функцию, которая подсчитывает количество цифр в данном тексте.

5.3.11. Программа. В стандартном потоке ввода задается последовательность ненулевых целых чисел, за которой следует 0. Вывести сначала все отрицательные числа этой последовательности, а затем – все положительные (в любом порядке).

5.3.12. На стандартном потоке ввода задана формула следующего вида:

<формула> ::= <цифра> | (<формула> <знак> <формула>)
*<знак> ::= + | - | **
<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Считая, что в записи формулы нет ошибок, описать функцию, вычисляющую значение этой формулы.

6. Массивы

6.1. Одномерные массивы, адресная арифметика

Одномерные массивы. Определение и инициализация массива, доступ к элементам массива. Указатели и массивы, адресная арифметика. Передача массива в качестве параметра в функцию. Задачи.

6.1.1. Одномерные массивы

Определение одномерного массива имеет вид:

```
<тип> <имя> [<количество элементов массива>]
```

, где <тип> задает тип элементов массива, при этом важно помнить, что все элементы массива имеют один и тот же тип. При размещении в памяти все элементы массива располагаются подряд. Доступ к элементу массива осуществляется по его индексу, в качестве индекса используются целые неотрицательные числа, индексация элементов массива начинается с 0. Таким образом, в массиве из `n` элементов индексация элементов изменяется в диапазоне от `0` до `n-1`.

Например, определение целочисленного массива `x` из трех элементов:

```
int x[3]; // элементы x[0], x[1], x[2]
```

При определении массива возможна его полная или частичная инициализация начальными значениями:

```
int x[3] = {1, 2, 3}; // x[0] = 1, x[1] = 2, x[2] = 3
int a[10] = {1, 2, 3}; // a[0] = 1, a[1] = 2, a[2] = 3
int b[10] = {[0] = 1, [3] = 2, [5] = -3};
 // b[0] = 1, b[3] = 2, b[5] = -3
```

Не указанные в инициализации элементы массива в общем случае не получают никакого начального значения. Исключение составляют глобальные массивы, все элементы которых по умолчанию инициализируются нулевыми значениями при определении массива.

Количество элементов массива может объявляться с помощью константы, в этом случае рекомендуется использовать конструкцию `enum`. Например:

```
enum {BUF_SIZE = 100}; // объявление константы BUF_SIZE
int buffer[BUF_SIZE];
```

Выход за границу массива обычно не контролируется компилятором, а обнаруживается только на этапе выполнения программы. Во избежание подобной ошибки программист должен контролировать данную ситуацию самостоятельно.

Во всех приведенных выше примерах размер массива задается константой, однако стандарт Си-99 допускает также использование массивов переменной длины в качестве локальных массивов. Например, ввод целочисленного массива из `n` элементов в стандарте Си-99 реализуется следующим фрагментом:

```
int n;
```

```
scanf ("%d", &n);// ввод количества элементов массива
int x[n];
for (int i =0; i < n; i++) {// ввод массива
  scanf ("%d", &x[i]);
}
```

Задача 1. Укажите, какому элементу массива будет присвоено значение `12`, а также значение переменной `i` после выполнения следующего фрагмента программы:

```
int i = 8;
a[i++] = 12;
```

Решение: постфиксная операция `++` увеличит значение переменной `i` *после* ее использования в выражении. Следовательно, ответ: `a[8] = 12, i = 9.`

*Задача 2***.** На стандартном потоке ввода задан текст, состоящий из латинских букв и цифр и оканчивающийся точкой. На стандартный поток вывода вывести цифру, наиболее часто встречающуюся в тексте (если таких цифр несколько, вывести любую из них).

Для решения задачи сформируем массив счетчиков вхождений цифр в заданный текст, где в `i`-ом элементе массива будем накапливать количество вхождений в текст цифры со значением `i`. Если символьная переменная `c` содержит цифру, тогда выражение `c - '0'` вычисляет целочисленное значение этой цифры и, следовательно, индекс соответствующего счетчика в массиве. Сформировав массив счетчиков, найдем индекс (`ind_max`) максимального элемента в этом массиве. Тогда выражение `ind_max + '0'` и будет определять искомую цифру.

```
#include <stdio.h>
int
main (void)
{
  int digits[10]; // массив счетчиков
  // инициализация массива счетчиков
  for (int i =0; i < 10; i++) {
    digits[i] = 0;
  }
  // ввод текста и формирование массива счетчиков
  char c; // очередной символ
  while ((c = getchar ()) != '.') {
    if (c >= '0' && c <= '9') {
      ++digits[c - '0'];
    }
  }
  // поиск индекса максимального элемента
  int max = digits[0], ind_max = 0;
  for (int i =1; i < 10; i++) {
    if (digits[i] > max) {
      max = digits[i];
      ind_max = i;
    }
  }
  printf ("%c\n", ind_max + '0');
  return 0;
}
```

6.1.2. Указатели и массивы

Имя массива является константным указателем на нулевой элемент массива, т.е. имеет тип:

```
int * const;
```

Изменять значение такого указателя нельзя, например, ошибкой было бы написать следующее присваивание:

```
int a[10];
int b[10];
a = b; // ЗАПРЕЩЕНО!  – попытка изменить значение константы a
```

Однако можно установить обычный (неконстантный) указатель на начало массива и затем изменять его значение по своему усмотрению, например:

```
int a[10];
int *p;
p = a;
```

В результате указатель `p` будет указывать на начало (нулевой элемент) массива `a`.

Для вычисления размера памяти (в байтах), необходимой для хранения объекта, используется операция `sizeof`. Так, значением выражения `sizeof(<тип>)` является количество байтов, необходимых для размещения в памяти значения указанного типа, выражение `sizeof(a)`, где `a` – имя некоторой переменной, возвращает количество байтов памяти, отводимых компилятором для хранения переменной `a`. Для массива `a` из `n` элементов `sizeof(a) = sizeof(<тип>) * n`, где `<тип>` задает тип элементов массива. Например, при условии, что для хранения значения типа `int` отводится 4 байта памяти, для массива `a` из приведенного выше примера

```
sizeof(a) = sizeof(int) * 10 = 4 * 10 = 40,
```

Обратите внимание, что `sizeof(a)` ≠ `sizeof(p)`, так как результатом вычисления выражения `sizeof(p)` является размер памяти, отводимой компилятором под указатель `p` (под объект `int *)`, обычно это 4 байта.

6.1.3. Адресная арифметика

Над указателями определены следующие основные операции:

- добавление и вычитание целочисленной константы,
- сравнение (==, !=, <, >, <=, >=),
- вычитание указателей одного типа.

Например:

```
int a[10];
int *p, x;
p = a;
x = *(p + 8); // x = a[8]
```

При вычислении выражения `p + 8` адрес, хранящийся в указателе `p`, складывается со значением `8*sizeof(int)`, в результате получаем адрес восьмого элемента массива `(&a[8])`. После выполнения операции разыменования `*(p + 8)` в переменной `x` будет значение восьмого элемента массива `(a[8])`. Обратите внимание на необходимость использования скобок в выражении `*(p + 8)`. Сравните:

```
x = *p + 8; // x = a[0] + 8
x = *(p + 8); // x = a[8]
```

Аналогично выражение `p - i`, где `i` – некоторая целочисленная константа, задает указатель на `i` -ый перед `p` элемент массива.

При выполнении операций `+=`, `-=`, `++`, `--` указатель передвигается на соответствующее количество элементов массива. Например,

```
p += 8; // p = &a[8]
```

при условии, что перед выполнением операции `p` указывал на начало массива `a`.

Указатели одинакового типа можно сравнивать и вычитать. Эти операции имеет смысл выполнять при условии, что оба указателя указывают на элементы одного и того же массива. Так, если указатели `p` и `q` указывают на элементы одного и того же массива и при этом номер элемента, на который указывает `p`, меньше номера элемента с указателем `q`, то справедливо соотношение:

`p < q.`

При вычитании указателей вычисляется разность адресов, хранящихся в этих указателях, которая затем делится на размер (в байтах) элемента массива. Результатом является количество элементов массива, расположенных между данными указателями. Например:

```
int a[10];
int *p, *q;
p = a;
q = &a[8];
// q – p = (&a[8] - &a[0]) / sizeof(int) = 8
```

Задача 3. Что будет напечатано в результате выполнения следующего фрагмента:

```
int a[3] = {1, 2, 3};
int *px, x = 5;
px = a + 1;
*px-- = x – 10;
printf("%d\n", *px);
printf("%d %d %d\n", a[0], a[1], a[2]);
```

Рассмотрим выполнение присваиваний из данного фрагмента.

```
px = a + 1; // px = &a[1]
*px-- = x – 10; // a[1] = -5, px = &a[0]
```

Особый интерес представляет выполнение второго присваивания, где сначала вычисляется значение выражения в правой части присваивания `(x – 10 = -5)`, затем это значение присваивается по указателю `px (a[1] = -5)`, и указатель передвигается

на предыдущий элемент массива (`px = &a[0]`). Таким образом, в результате выполнения указанного фрагмента будет напечатано:

```
1
1 -5 3
```

6.1.4. Передача массива в функцию

При передаче массива в функцию используется то, что имя массива является константным указателем на его нулевой элемент: при указании в качестве параметра имени массива в функцию фактически передается по значению этот указатель (т.е. копия адреса начала той области памяти, в которой находятся элементы массива). Из типа указателя также известно, сколько места занимает каждый элемент массива, что позволяет обратиться к любому элементу массива. Но информация о количестве элементов массива теряется, поэтому размер массива следует передавать в функцию через дополнительный параметр. Например, рассмотрим функцию вычисления суммы элементов целочисленного массива из `n` элементов. Ниже приведены два эквивалентных описания прототипа этой функции:

```
int sum(int t[], int n);
```

Здесь `t` – имя массива, при этом наличие квадратных скобок обязательно.

```
int sum (int *t, int n);
```

Здесь `t` – указатель на целое (на элемент массива с индексом 0).

Вызов функции:

```
int a[10];
int x, y, z;
…
x = sum(a, 10); // сумма элементов всего массива
y = sum(a, 5); // сумма элементов первой половины массива
z = sum(a + 5, 5); // сумма элементов второй половины массива
```

Задача 4. Описать функцию вычисления скалярного произведения двух вещественных массивов из `n` элементов.

```
double
scalar (double x[], double y[], int n) {
  double s = 0.0;
  int i;

  for (i = 0; i < n; i++) {
    s += x[i] * y[i];
  }
  return s;
}
```

6.2. Строки

Понятие строки. Строковые константы (строковые литералы). Ввод и вывод строк. Работа со строками. Задачи.

6.2.1. Строки и строковые константы

Под строкой в языке Си понимается последовательность символов, оканчивающаяся символом с кодом 0 (`'\0'`). В языке нет специального строкового типа, для хранения строк используются массивы типа `char`, `signed char`, `unsigned char` . Поэтому строки, по сути, являются массивами символов, отличающимися от обычных массивов обязательным наличием символа `'\0'` в конце строки. Любая программа обработки строк определяет конец строки именно по данному символу (`'\0'`). Кроме этой особенности, можно отметить следующие специфические особенности строк:

- строка не может содержать внутри себя символ с кодом 0,
- в языке Си не предусмотрено ограничений на длину строки,
- длина строки не содержится в строке (в отличие от других языков программирования, где есть понятие строки переменной длины).

Определение строки оформляется как определение символьного массива соответствующего типа, например:

```
char s[N];// строка из N символов
```

Максимальное количество значащих символов такой строки равно `N - 1` , поскольку в строке всегда должен присутствовать символ `'\0'`.

При определении строки по аналогии с массивами можно задать ее инициализацию. Для инициализации строк используются строковые константы (строковые литералы) вида `"<последовательность символов>"`. Такая константа имеет тип `char const *` и в своем внутреннем представлении заканчивается символом `'\0'` , поэтому фактическая длина константной строки в памяти всегда на 1 больше, чем количество символов, указанных в двойных кавычках. Например, инициализация:

```
char er[10]= "error";// er[0]='e', …, er[4] = 'r', er[5]= '\0'
```

определяет строку из 10 символов, первые пять из которых получат указанные в строковой константе значения, а шестой будет содержать символ с кодом 0. Как и в случае символьных констант, префикс L позволяет задать строковую константу в кодировке Unicode, например, `L"unicode string"`.

Для ввода/вывода строк можно использовать функции `scanf` и `printf` со спецификатором формата `%s` . Например:

```
char str[10];
scanf ("%s", str);// ввод строки
```

Функция `scanf` последовательно считывает в строку `str` символы вводимой строки (предварительно пропустив пробелы, символы перевода на следующую строку и т.п.) и

автоматически добавляет в конец строки символ '\0'. Ввод выполняется до первого символа пробел, символа перевода на следующую строку и т.п. (полный перечень таких символов см. в описании функции scanf [1]). Размер массива str должен быть достаточным для размещения в нем вводимой строки и добавленного к ней символа '\0'. Существенным недостатком данного варианта функции scanf, является то, что здесь никак не контролируется количество вводимых символов, что может представлять угрозу для работы программы. Поэтому при использовании функции scanf рекомендуется задавать ограничение на размер вводимой строки:

```
char str[10];
scanf ("%9s", str);//ввод <= 9 символов с добавлением '\0'
            // в конец строки
```

Вывод строки:

```
char str[10];
printf ("%s", str);
```

При выводе строки на печать выводятся все значащие символы строки до символа '\0'.

6.2.2. Работа со строками

При работе со строками используется стандартная библиотека функций. Подключить ее можно директивой препроцессора:

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

В качестве примеров работы со строками рассмотрим реализацию некоторых функций из этой библиотеки. При обработке строк удобно использовать указатели, что и будет продемонстрировано в приведенных ниже примерах.

Задача 1. Реализовать функцию, которая возвращает длину (количество значащих символов) строки cs, с прототипом:

```
size_t strlen (const char *cs);
```

где size_t – это тип, который используется в стандартной библиотеке Си для задания размеров объекта и является синонимом (typedef) некоторого целого типа (часто – unsigned long).

Для решения задачи воспользуемся двумя указателями, один из которых (cs) будет указывать на начало строки, а другой (tmp) будем перемещать по строке до первого символа с кодом 0. Разность указателей и даст результат функции.

```
size_t
strlen (const char *cs)
{
  char *tmp = cs;
  while (*tmp) // пока очередной символ ≠ '\0'
    tmp++;
  return tmp - cs;
}
```

Задача 2. Реализовать функцию с прототипом:

```
char *strcpy (char *dst, const char *src);
```

для копирования строки src (включая '\0') в строку dst . Функция возвращает указатель на первый символ строки dst.

```
char *
strcpy (char *dst, const char *src)
{
  char *tmp = dst;
  while (*dst++ = *src++)
    ;
  return tmp;
}
```

Основной интерес в предложенной реализации представляет выражение в условии цикла. Оно вычисляется следующим образом: берется символ по указателю src (после чего указатель src передвигается на следующий символ копируемой строки) и помещается по указателю dst, после чего указатель dst аналогично передвигается на следующий символ строки. Значением выражения присваивания является только что скопированный символ, который сравнивается с '\0' для контроля завершения цикла. Таким образом, в результате выполнения цикла в строку dst копируются все символы исходной строки src, включая завершающий символ '\0'. Обратите внимание на порядок выполнения операций при вычислении выражения *src++ . Операции * и ++ имеют одинаковый приоритет и выполняются справа налево. Однако постфиксная операция ++ не изменяет указатель src, пока по нему не получено значение. В итоге значением данного выражения является символ, на который указывал указатель src до своего инкрементирования.

Заметим, что библиотечная функция strcpy никак не контролирует возможность переполнения массива. Контроль переполнения массива в языке Си обязан осуществлять сам программист. В связи с этим рекомендуется использовать более надежный и безопасный вариант функции копирования (strncpy) с прототипом:

```
char *strncpy (char *dst, const char *src, size_t n);
```

в котором предусмотрена возможность ограничить размер копируемой строки. Параметр n задает максимальное число копируемых символов. Данная функция копирует не более n символов строки src в строку dst, при этом результирующая строка дополняется символами '\0', если в строке src меньше n символов. Функция возвращает указатель на первый символ результирующей строки (dst). Обратите внимание, что если длина (количество значащих символов) исходной строки src больше или равна n, то копируются первые n символов строки и символ '\0' при этом не добавляется.

Данный «безопасный» вариант функции копирования (strncpy) по сравнению с «опасным» вариантом (strcpy) при своей работе выполняет дополнительные операции – сравнение количества скопированных символов с параметром n, что приводит к значительным дополнительным затратам, поскольку сама функция копирования очень проста. Поэтому нужно делать осознанный выбор между опасным, но более

производительным вариантом и безопасным, но более медленным. Например, использовать безопасный вариант `strncpy` только при начальной проверке ввода пользователя на корректность, а далее проверку выхода за границы буфера выполнять самостоятельно пред вызовом функции `strcpy`.

Приведем краткое описание еще нескольких функций стандартной библиотеки для работы со строками.

1) `char *strcat (char *s, const char *ct);`

Функция приписывает строку `ct` в конец строки `s`; возвращается указатель на первый символ строки `s`.

Так же, как для функции копирования, существует более надежный вариант этой функции с прототипом:

```
char *strncat (char *s, const char *ct, size_t n);
```

, который ограничивает (<= `n`) количество приписываемых символов. Именно этот вариант и рекомендуется использовать для конкатенации строк.

2) `int strcmp(const char *cs, const char *ct);`

Функция сравнивает в лексикографическом порядке строку `cs` со строкой `ct`. Если строка `cs` меньше строки `ct`, возвращается значение < 0, если строка `cs` больше строки `ct`, возвращается значение > 0, в случае равенства строк возвращается значение 0.

Обратите внимание, что для сравнения строк нельзя использовать операции `==`, `!=` и т.п., поскольку при этом происходит сравнение указателей (адресов) на начало соответствующих строк, а не самих строк.

3) `char *strchr (const char *cs, char c);`

Функция возвращает указатель на первое вхождение символа `c` в строку `cs` или `NULL`, если такого не оказалось.

4) `char *strrchr (const char *cs, char c);`

Функция возвращает указатель на последнее вхождение символа `c` в строку `cs` или `NULL`, если такого не оказалось.

6.3. Двумерные массивы (матрицы)

Многомерные массивы. Определение и инициализация двумерного массива (матрицы), доступ к элементам матрицы. Передача матрицы в качестве параметра в функцию. VLA-массивы. Задачи.

6.3.1. Двумерные массивы и работа с ними

В языке Си можно определять многомерные массивы, при этом массив размерности `n` рассматривается как одномерный массив, каждый элемент которого представляет собой массив размерности `n - 1`. Так, двумерный массив является фактически одномерным массивом, каждый элемент которого в свою очередь также является одномерным массивом. Далее будем рассматривать двумерные массивы (матрицы).

Определение двумерного массива (матрицы) имеет вид:

```
<тип> <имя> [<количество строк>] [<количество столбцов>]
```

, где <тип> задает тип элементов массива. При размещении в памяти элементы матрицы располагаются по строкам. Пример определения матрицы с целочисленными элементами:

```
int matrix[5][10]; // матрица из 5 строк и 10 столбцов
```

Здесь матрица matrix определяется как одномерный массив из 5 элементов, каждый из которых в свою очередь является массивом из 10 целочисленных элементов.

При определении матрицы так же, как и при определении одномерного массива, возможна ее полная или частичная инициализация начальными значениями, при этом каждая строка матрицы инициализируется соответствующим вложенным подсписком:

```
int matrix[3][2] =
{
  {4, 5},  // 0-я строка матрицы
  {-1, 1}, // 1-я строка матрицы
  {0, -7}  // 2-я строка матрицы
}
```

Доступ к элементам матрицы matrix осуществляется с помощью выражения matrix[i][j], где i – номер строки и j – номер столбца матрицы.

Задача 1. Напишите фрагмент программы для вычисления следа квадратной вещественной матрицы n – го порядка (n = 20).

```
enum {N = 20};
double matr[N][N];
double s = 0.0; // след матрицы
int i;
for (i =0; i < N; i++) {
  s += matr[i][i];
}
```

При работе с двумерным массивом можно использовать указатель на массив. Например:

```
int matrix[5][10]; // определение двумерного массива (матрицы)
int (*pmatr)[10];  // определение указателя на массив
```

Здесь указатель pmatr определяется как указатель на массив из 10 целочисленных элементов и, следовательно, ему может быть присвоен адрес любой строки матрицы. Заметим, что наличие круглых скобок в определении указателя на массив обязательно, поскольку операция [] имеет более высокий приоритет, чем операция *. Сравните:

```
int (*pmatr)[10];// определение указателя на массив
int *pmatr[10];  // определение массива из 10 указателей
                 // на значения типа int
```

Для иллюстрации работы с указателем на массив рассмотрим следующую задачу.

Задача 2. Что будет напечатано в результате выполнения данного фрагмента программы:

```
int m[3][3] =
{
  {1, 3, 5},
  {7, 9, 11},
  {13, 15, 17}
};
int *p1;
int (*p2)[3];
p1 = m[1];
p2 = m + 1;
p1++;
p2++;
printf ("%d %d\n", *p1, **p2)
```

Для решения задачи рассмотрим схему доступа к элементам матрицы m.

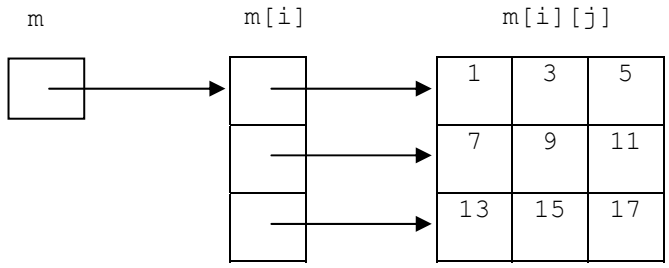

Рис. 4. Доступ к элементам матрицы

Указателю p1, как указателю на целочисленное значение, может быть присвоен адрес любого элемента матрицы m. Так, в результате выполнения присваивания

```
p1 = m[1];
```

p1 будет присвоен адрес 0-го элемента 1-й строки матрицы (m[1][0]). Заметим, что такое присваивание эквивалентно присваиванию

```
p1 = &m[1][0];
```

Указатель p2 – это указатель на массив из трех целочисленных элементов, следовательно, ему может быть присвоен адрес любой строки матрицы m. В результате выполнения присваивания

```
p2 = m + 1;
```

в p2 будет зафиксирован адрес 1-ой строки матрицы m. Тогда оператор p1++; передвинет указатель p1 на следующий элемент 1-ой строки матрицы, и *p1 = m[1][1] = 9, а оператор p2++; передвинет указатель p2 на следующую строку

матрицы, и `**p2 = m[2][0] = 13`. Следовательно, в результате выполнения указанного фрагмента будет напечатано:

```
9 13
```

При передаче матрицы в качестве параметра в функцию необходимо указать количество столбцов матрицы, количество строк при этом необязательно, поскольку фактически передается указатель на массив строк (при необходимости количество строк может передаваться с помощью дополнительного параметра). Ниже приведен пример трех эквивалентных описаний прототипа функции, параметром которой является двумерный массив (матрица):

```
int f(int matr[5][10]);
int f(int matr[][10]);
int f(int (*matr)[10]);
```

Как видим, в данных описаниях размер второго измерения матрицы фиксирован и не может быть изменен. Причиной этого является допущение в стандарте Си-89 только константных размеров массивов. Функция `f` из описания выше не может быть использована, например, для массива `int a[7][7]`. Поэтому в реальных программах многомерные массивы описывались и выделялись в памяти как одномерные, а вычисление индекса необходимого элемента ложилось на программиста.

В Си-99 возможно описывать локальные многомерные массивы неконстантного размера (т.н. VLA-массивы), а также определять указатели на такие массивы и, как следствие, выделять для них динамическую память. Без использования указателей VLA-массивы могут быть только переменными автоматического класса памяти, то есть использоваться как локальные переменные либо параметры функций. Важно, что после создания VLA-массива его размер больше не может быть изменен. Использование VLA-массивов иллюстрируют следующие примеры:

```
// VLA-массив как локальная переменная
int foo (int n)
{
  int a[n];
  <... Можно обрабатывать a[i]...>
}

// VLA-массив как параметр функции
// Можно передать двумерную матрицу
// int a[???][???] любого размера
int asum2d (int m, int n, int a[m][n])
{
  int s = 0;
  for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
      s += a[i][j];
  return s;
}
```

Если VLA-массив используется в качестве параметра функции и размерности массива также являются параметрами, то его описание должно следовать после описания

параметров. Объявление в списке параметров VLA-массива до его размерностей не допускается. При объявлении функции, содержащей VLA-массив как параметр, можно опустить имя массива, как и для обычных параметров, но при этом необходимо на месте размерностей массива указать звездочки для того, чтобы отличить его от обычного массива:

```
/* объявление функции asum2d */
int asum2d (int, int, int [*][*]);
```

Необходимо отметить, что как и для обычного массива, выделение памяти под VLA-массив, заданный как параметр функции, не производится: передается указатель на первый элемент массива, но при этом компилятору известен его размер по обоим измерениям, что позволяет использовать индексацию в теле функции.

Кроме создания VLA-массива в автоматической памяти, возможно выделение под него динамической памяти (работа с динамической памятью рассматривается в разделе 7.3). В таком случае, например, для двумерного массива определяется указатель на VLA-массив из `n` элементов, под который можно выделить динамическую память на `m × n` элементов. Как и в случае параметра функции, компилятор обеспечит корректную адресацию элементов при индексации по обоим измерениям:

```
int main (void)
{
  int m, n;
  scanf ("%d%d", &m, &n);
  int (*pa)[n];
  pa = (int (*)[n]) malloc (m * n * sizeof (int));
     <... Считываем pa[i][j]...>
  s = asum2d (m, n, pa);
  free (pa);
}
```

Задача 3. Описать функцию для вывода на печать целочисленной матрицы из `m` строк и `n` столбцов.

Приведем две реализации предложенной функции. В первой из них матрица рассматривается как единая область памяти, т.е. одномерный массив из `m × n` элементов, во второй как VLA-массив.

```
// (1) построчный вывод матрицы как единой области памяти
void
print_matr (int m, int n, int *a)
{
  int i, j;
  for (i = 0; i < m; i++){
    for (j = 0; j < n; j++){
      printf ("%d ", a[i * n + j]);// вывод a[i][j]
    }
    printf ("\n");
  }
}
```

Описанная таким образом функция `print_matr` будет использоваться следующим образом:

```
int m, n;
if (scanf ("%d%d", &m, &n) == 2) {
  int *matrix = malloc (m * n * sizeof (int)); /* выделение
  динамической памяти под матрицу */
  // заполнение матрицы значениями
  for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
      matrix[i * n + j] = (i + j) / 2;
  print_matr (m, n, matrix);
}
```

```
// (2) построчный вывод матрицы как VLA-массива
void
print_matr (int m, int n, int a[m][n])
{
  int i, j;
  for (i = 0; i < m; i++){
    for (j = 0; j < n; j++){
      printf ("%d ", a[i][j]);
    }
    printf ("\n");
  }
}
```

В этом случае возможно такое использование функции `print_matr`:

```
int m, n;
if (scanf ("%d%d", &m, &n) == 2) {
  int matrix[m][n]; // или в динамической памяти
  // заполнение матрицы значениями
  for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
      matrix[i * n + j] = (i + j) / 2;
  print_matr (m, n, matrix);
}
```

6.4. Задачи для самостоятельного решения

Массивы

6.4.1. Описать функцию, которая проверяет, обладает ли целочисленный массив из n элементов указанным свойством. В случае положительного ответа функция возвращает значение 1 и 0 в противном случае. Свойства:

a) элементы массива упорядочены по возрастанию;
b) массив является симметричным (равноудаленные от концов массива элементы равны друг другу).

6.4.2. Описать функцию, определяющую индекс первого по порядку элемента целочисленного массива из n элементов, значение которого равно заданному числу x. В

случае отсутствия в массиве указанного значения функция возвращает -1. При решении задачи рассмотреть варианты:

a) массив произвольный,
b) элементы массива упорядочены по возрастанию (использовать алгоритм двоичного поиска).

6.4.3. Описать функцию, которая в целочисленном массиве из n элементов меняет местами максимальный и минимальный элементы (считать, что в массиве все элементы различные).

6.4.4. Описать функцию, осуществляющую циклический сдвиг на m позиций вправо элементов вещественного массива из n элементов ($m < n$).

6.4.5. Описать функцию, которая упорядочивает по неубыванию элементы целочисленного массива из n элементов, используя следующий алгоритм сортировки:

a) сортировка выбором: находится максимальный элемент массива и меняется местами с последним элементом, затем тот же метод применяется ко всем элементам массива, кроме последнего (он уже находится на своем окончательном месте), и т.д.
b) сортировка обменом (метод пузырька): последовательно сравниваются пары соседних элементов массива x_k и x_{k+1} ($k = 0, 1, \ldots, n - 2$) и, если $x_k > x_{k+1}$, то они переставляются; в результате наибольший элемент окажется на своем месте в конце массива; затем тот же метод применяется ко всем элементам массива, кроме последнего, и т.д.

6.4.6. Описать функцию, которая для целочисленного массива из n элементов определяет:

a) присутствуют ли в массиве одинаковые элементы (в случае положительного ответа возвращается значение 1 и 0 в противном случае),
b) количество различных элементов в массиве.

6.4.7. Решить предыдущую задачу в предположении, что значения элементов массива находятся в диапазоне [-100, 100].

Строки

Указание: при решении задач не использовать стандартную библиотеку работы со строками.

6.4.8. Привести реализацию описанных выше функций `(1) – (4)`.

6.4.9. Описать функцию, которая изменяет заданную строку следующим образом: сначала записывает в нее все элементы с четными индексами, а затем все элементы с нечетными индексами (с сохранением их относительного порядка в каждой группе).

Например: `abcdefgh => acegbdfh, vwxyz => vxzwy.`

6.4.10. Описать функцию, которая в заданной строке меняет местами ее первую и вторую половины.

Например: `abcdefgh => efghabcd, vwxyz => yzxvw.`

6.4.11. Описать функцию, «переворачивающую» содержимое строки.

Например: `abc => cba.`

6.4.12. Описать функцию, получающую в качестве параметров две строки и отвечающую на вопрос, является ли первая строка префиксом второй.

6.4.13. Описать функцию, получающую в качестве параметров две строки и отвечающую на вопрос, является ли первая строка суффиксом второй.

Двумерные массивы (матрицы)

6.4.14. Описать функцию, которая в целочисленной матрице из n строк и m столбцов меняет местами максимальный и минимальный элементы матрицы (считать, что все элементы матрицы различны).

6.4.15. Описать функцию, которая в вещественной матрице из n строк и m столбцов определяет номер строки, содержащей максимальное количество положительных элементов (считать, что такая строка только одна).

6.4.16. Описать функцию, которая находит произведение двух вещественных матриц порядка n × m и m × k, соответственно.

6.4.17. Описать функцию, которая для квадратной целочисленной матрицы n-го порядка определяет, является ли данная матрица:

a) симметричной относительно главной диагонали;
b) магическим квадратом (суммы элементов во всех строках и столбцах одинаковы).

В случае положительного ответа функция возвращает значение 1 и 0 в противном случае.

6.4.18. Описать функцию, которая в целочисленной матрице из n строк и m столбцов подсчитывает количество строк:

a) нулевых (все элементы строки равны нулю);
b) все элементы которых имеют одинаковый знак (считать, что все элементы матрицы отличны от 0);
c) упорядоченных по возрастанию.

7. Динамическая память. Размещение массивов в динамической памяти. Массивы указателей

Динамическая память. Функции работы с динамической памятью. Размещение массивов в динамической памяти. Массивы указателей. Размещение матрицы в динамической памяти. Задачи.

7.1. Динамическая память. Функции работы с динамической памятью

В тех случаях, когда размер входных данных программы заранее не известен, для их размещения используется динамическая память. Динамическая память – это специальная область памяти (куча), выделение и освобождение памяти в которой происходит динамически по требованию программиста.

Для работы с динамической памятью используется стандартная библиотека функций. Подключить ее можно директивой препроцессора:

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

Рассмотрим основные стандартные функции для работы с динамической памятью.

Для выделения динамической памяти предусмотрены функции malloc и calloc.

```
void *malloc (size_t n);
```

Функция malloc выделяет в куче область памяти размера n байтов и возвращает указатель на начало этой области или NULL, если выделить память не удалось. Выделенная область памяти не инициализируется.

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

Функция calloc выделяет в куче область памяти для размещения n объектов размера size и возвращает указатель на начало этой области или NULL, если выделить память не удалось. Выделенная область памяти инициализируется нулями.
Для корректной работы программы при использовании данных функций всегда следует проверять возвращаемый ими результат.

Функции malloc и calloc возвращают указатель обобщенного типа (void *). При присвоении такого указателя указателю на конкретный тип будет выполнено неявное приведение типов, например:

```
char *p = calloc (n, sizeof(char));/* неявное приведение
                                      указателя void *к char * */
```

Тем не менее, иногда программисты используют явное приведение типа: оно могло быть необходимо, если код создавался для старых версий языка Си (до стандарта Си-89), либо для последующего использования кода также и для языка Си++, в котором требуется явное приведение.

В качестве примера размещения данных в динамической памяти рассмотрим следующую задачу.

Задача 1. Написать фрагмент программы, в котором из стандартного потока ввода вводится строка длиной не более 100 символов и размещается в динамической памяти.

```
char str[101], *p;
if (fgets (str, 100, stdin) != NULL){/* ввод строки из
                                        стандартного потока ввода stdin */
  p = malloc (strlen(str) + 1);
  if (p) {//выделение памяти выполнено успешно
    strcpy (p, str);
  }
}
```

Обратите внимание на то, что в динамической памяти строка занимает ровно столько места, сколько реально требуется для ее размещения. Дальнейший доступ к элементам строки может осуществляться как по указателю, так и с помощью операции индексирования.

Часто требуется увеличить размер памяти, выделенный под конкретный указатель, например, если заполнился некоторый буфер. В таком случае используется функция:

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

Функция realloc изменяет размер выделенной области до size байт (чаще всего используется для увеличения) и возвращает указатель на новую область или NULL, если выделить память указанного размера не удалось. Содержимое области от начала до меньшего из старого и нового размеров не изменяется. Указатель ptr должен быть ранее сформирован функциями malloc, calloc или realloc. Если этот указатель равен

NULL, то вызов эквивалентен malloc, а если размер равен нулю – то вызову free. Часто удается увеличить область без изменения самого указателя на область (т.е. ее адреса), в таком случае функцией возвращается то же самое значение указателя. Но на это нельзя полагаться и всегда нужно обновлять значение указателя, например:

```
p = realloc (p, sizeof (int) * new_size);
```

Функцию realloc не рекомендуется часто вызывать, поскольку она достаточно "затратная" по времени выполнения. Если ее использование необходимо, то при вызове, когда неизвестно точно сколько требуется памяти - рекомендуется удваивать занятую память.

Освобождение динамической памяти выполняет функция:

```
void free (void *p);
```

Функция освобождает в куче область памяти, на которую указывает p (если p == NULL, функция ничего не делает). Обратите внимание на тот факт, что размер освобождаемой области памяти не надо передавать в функцию, поскольку функция free автоматически извлекает его из служебной информации, сформированной функциями выделения памяти и хранящейся непосредственно перед данной областью. Поэтому для корректного освобождения памяти указатель p должен указывать на область памяти, ранее выделенную одной из функций malloc, calloc или realloc. В противном случае результат работы функции не определен.

При работе с динамической памятью всегда следует соблюдать правило освобождать выделенную ранее память, если отпала необходимость в ее использовании, чтобы дать возможность библиотеке снова выделить эту память под последующие запросы. Иначе возникают т.н. *утечки памяти*, которые приводят к невозможности выделить динамическую память, когда на самом деле часть ее не используется. Если приложение работает длительное время, то оно может занять всю свободную память.

В качестве примера обработки массива, размещенного в динамической памяти, рассмотрим следующую задачу.

Задача 2. На стандартном потоке ввода задается целое число n и массив из n вещественных чисел. Используя для размещения массива динамическую память, найти сумму элементов массива, значения которых меньше значения последнего элемента массива.

```
#include <stdio.h>
#include <stdlib.h>
int
main(void)
{
  int i, n;
  double *x;
  double s = 0.0;
  scanf("%d", &n);
  x = malloc (n * sizeof (double)); // выделение памяти под массив
  if (!x){        // выделить память не удалось
    fprintf(stderr, "out of memory\n");
    return 1;
  }
  // ввод и обработка массива
```

```
  for (i =0; i < n; i++) {
    scanf("%lf", &x[i]);
  }
  for (i =0; i < n - 1; i++) {
    if (x[i] < x[n - 1]) s += x[i];
  }
  printf ("%f\n", s);
  free (x);// освобождение памяти от массива
  return 0;
}
```

7.2. Массивы указателей

В массиве указателей все элементы массива представляют собой указатели на объекты некоторого (единого для всех элементов массива) типа. Например, определение массива из 10 указателей на объекты типа `int`:

```
int *p[10];
```

Наиболее часто массивы указателей применяют для организации хранения строк, когда каждый элемент массива представляет собой указатель на соответствующую строку, т.е. содержит адрес начала этой строки. Преимуществом использования массива указателей в данном случае является то, что строки могут иметь различную длину.

Рассмотрим массив указателей на строки. Начальную инициализацию такого массива проиллюстрируем на примере массива указателей на строковые константы.

```
const char *ptr[] = {"red", "yellow", "green", NULL};
```

Элементами данного массива являются адреса строковых констант, последний элемент массива инициализируется нулевым указателем `NULL` (см. рисунок).

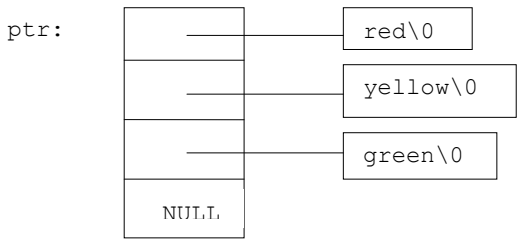

Рис. 5. Массив указателей на строки

Задача 3. Описать функцию, которой в качестве параметров передается массив указателей на строки (признак конца массива – указатель `NULL`) и символ. Функция возвращает количество строк, начинающихся с указанного символа.

```
int
numbstr (char *s[], char ch)
{
  int i = 0, n = 0;
  while (s[i] != NULL){
    if (*s[i] == ch) n++;
```

```
    }
    return n;
  }
```

7.3. Размещение матрицы в динамической памяти

Рассмотрим следующую задачу.

Задача 4. Описать функцию, которая выделяет в динамической памяти место под целочисленную матрицу размера `n × m`. Функция возвращает указатель на матрицу или `NULL`, если выделить память нужного размера не удалось.

Как уже упоминалось, в стандарте Си-89 для этого обычно выделялся сплошной участок памяти, а программист сам адресовал нужные элементы матрицы:

```
int *
matrix_create(int n, int m)
{
  return malloc (n * m * sizeof (int));
}

int *p = matrix_create (n, m);
// (i,j)-й элемент матрицы есть p[i*m+j]
```

В стандарте Си-99 можно задавать неконстантные размеры массивов. Тогда обработка матрицы, размещенной в динамической памяти, выглядит следующим образом:

```
void matrix_work (int n, int m) {
  int (*pm) matrix[m]; // указатель на массив из m элементов
  pm = malloc (sizeof (int) * n * m);
  pm[2][3] = 4;
  //можно передавать pm в функции
  // типа print_matr (см. раздел 6.3.1.)
  //с прототипами типа (int n, int m, int matrix[n][m])
}
```

При размещении матрицы в динамической памяти можно также использовать массив указателей, когда каждый элемент массива представляет собой указатель на соответствующую строку матрицы. Тогда сначала в динамической памяти следует выделить место под массив указателей на строки матрицы, а потом место непосредственно под элементы матрицы. Причем место в памяти под элементы матрицы может быть выделено как единым блоком, так и для каждой строки матрицы отдельно.

7.4. Задачи для самостоятельного решения

7.4.1. На стандартном потоке ввода задана последовательность строк, признаком конца последовательности является пустая строка. Количество строк не более 20. Написать фрагмент программы, который вводит строки, размещает их в динамической памяти и формирует массив указателей на эти строки. В качестве признака конца массива использовать нулевой указатель `NULL`.

7.4.2. Описать функцию, которой в качестве параметра передается массив указателей на строки (признак конца массива – указатель `NULL`):

a) функция выводит на стандартный поток вывода последний символ каждой строки;
b) функция выводит на стандартный поток вывода первые три символа каждой строки, длина которой больше или равна 3;
c) функция выводит на стандартный поток вывода строку максимальной длины;
d) функция подсчитывает количество строк, являющихся палиндромами;
e) функция упорядочивает строки в лексикографическом порядке.

7.4.3. Описать функцию, которая выделяет в динамической памяти место под целочисленную матрицу размера `n × m` при условии, что память под каждую строку матрицы выделяется отдельно. Функция возвращает указатель на матрицу или `NULL`, если выделить память нужного размера не удалось.

7.4.4. Описать функцию, которая освобождает в динамической памяти место, ранее выделенное под целочисленную матрицу размера `n × m`, при условии, что:

a) память под элементы матрицы выделялась единым блоком;
b) память под каждую строку матрицы выделялась отдельно.

7.4.5. На стандартном потоке ввода задано число `n (n > 2)` и элементы трех квадратных целочисленных матриц размера `n × n` . Используя для размещения матриц динамическую память, ввести эти матрицы и распечатать ту из них, в которой больше нулевых строк (считать, что такая матрица только одна).

8. Структуры и объединения.

Структуры. Описание типов структур и переменных этого типа. Операции над структурами. Объединения. Задачи.

8.1. Структуры

Структурный тип позволяет работать с совокупностью нескольких переменных как с единым целым. Переменные, являющиеся составными частями структуры, называются *полями*. Поля структуры могут быть любого типа.

Описание структурного типа выглядит следующим образом:

`struct` `[ <имя_типа_структуры> ] '{' <список_полей> '}' [ <переменные> ] ';'`

где <имя_типа_структуры> – идентификатор. В дальнейшем можно ссылаться на описанный тип при помощи конструкции вида **`struct`** <имя_структуры>. Этот идентификатор может быть опущен, в таком случае описываются только переменные анонимного структурного типа, а после описания завести новые переменные того же типа уже невозможно;

<список_полей> – одно или более описаний переменных, являющихся составными частями структуры. Эти описания имеют тот же синтаксис, что и описания обычных переменных в Си; при этом имена переменных в этих описаниях становятся именами соответствующих полей структуры. Поля могут быть любого типа, простого или составного, включая массивы и другие структуры. Поле структуры не может быть VLA-массивом;

<переменные> – список описываемых переменных, типом которых является только что объявленный структурный тип. Если этот список пуст, то описывается только структурный тип (в этом случае имя типа структуры не может быть опущено).

Имена полей структуры, а также имена типов структур имеют отдельные области видимости и могут совпадать с именами других объектов программы.

Описание структурного типа всегда описывает новый тип, не совпадающий ни с каким из ранее описанных (даже если список полей полностью совпадает со списком полей какого-либо другого структурного типа, описанным ранее).

Пример 1. Описание структурного типа для хранения информации о студенте:

```
struct student {
   char name[80]; // поле name – массив из 80 символов (строка)
   int group; // поле group – целое число
   char city[30]; // поле city – массив из 30 символов
} st1, st2; // переменные st1 и st2 имеют тип struct student

struct student course[400]; // массив из 400 структур типа
                            //  struct student
struct student *ps; // указатель на struct student
```

Пример 2. Описание структурного типа для хранения комплексного числа:

```
struct complex {
   double re, im;
}; /* непосредственно при описании типа структуры не описаны
      никакие переменные */
struct complex x, y; // x и y – переменные типа struct complex
```

Полями структуры, в свою очередь, могут быть другие структуры, то есть возможны вложенные структуры.

Над переменными структурного типа определены следующие операции:

1. Операция взятия адреса `( & )`.
2. Копирование и присваивание (для структур одного типа). При присваивании копируются значения всех полей структуры. Структуры можно передавать как параметры (по значению) в функцию, можно возвращать как результат функции.
3. Обращение к отдельному полю структуры. Для обращения используется оператор `'.'` (точка) в конструкции вида:

```
<переменная> . <имя_поля>
```

где `<переменная>` – переменная структурного типа. С полем структуры можно выполнять все действия, которые определены для типа, которому принадлежит это поле.

При обращении к полю структуры по указателю на структуру можно использовать оператор `'->'` (стрелка): `a -> b` эквивалентно `(*a).b`.

Приоритет операторов обращения к полям структуры (`.` и `->`) — наиболее высокий.

Примеры (с учетом приведенных выше описаний):

```
st1.group = 101; // значение поля group в перем. st1 равно 101
strcpy (st1.city, "Москва");
st2 = st1; // значения всех полей st1 копируются в поля st2
p = &st1;  // p указывает на st1
(*p).group++; /* увеличение поля group структуры, на которую
                 указывает p (т.е. St1) */
p->group++; // то же, что и в предыдущей строке

x.re += y.re;
x.im += y.im; // сложение комплексных чисел x и y
```

Никакие операции сравнения, а также арифметические или логические операции над структурами не определены:

```
if (x == y) … // ошибка!
            // Правильнее: if (x.re == y.re && x.im == y.im) .
x = x + y; // ошибка!
x++; // ошибка!
x.group = 102; // ошибка! В struct complex нет поля group.
st1 = x; // ошибка! Присваивание структур разных типов.
```

Оператор `sizeof` для структуры возвращает размер (в байтах) области памяти, отводимой компилятором под хранение структуры. Размер структуры в байтах всегда не меньше суммы размеров всех ее полей (может быть больше):

```
sizeof(struct complex)=sizeof(x) >= sizeof(double)+sizeof(double)= 16
```

При описании переменных-структур возможна их полная или частичная инициализация (как и в случае с массивами):

```
struct student st3 = {"Смирнов П.В.", 104, "Калуга"};
struct student st4 = {"Кузнецова А.Ю."};
```

Не указанные в инициализации поля структуры получают нулевое значение в случае, если явно инициализировано хотя бы одно поле структуры. Если инициализация вообще не производится, то, как и для обычных переменных, глобальные структуры получат нулевые значения, а локальные – не будут инициализированы по умолчанию.

Задача 1. Используя описанный выше тип `struct complex`, описать функцию для вычисления произведения двух комплексных чисел.

Возможны два варианта решения:

```
// Первый вариант: передача параметров по значению
struct complex mul (struct complex a, struct complex b) {
   struct complex m;
   m.re = a.re*b.re – a.im*b.im;
   m.im = a.re*b.im + a.im*b.re;
   return m;
}
```

```
// пример вызова:
c = mul (x,y);

// Второй вариант: передача параметров по указателю
struct complex mul (struct complex *a, struct complex *b) {
   struct complex m;
   m.re = a->re * b->re – a->im * b->im;
   m.im = a->re * b->im + a->im * b->re;
   return m;
}
// пример вызова:
c = mul (&x,&y);
```

Второй вариант решения более эффективен: при вызове такой функции не выполняется копирование содержимого параметров `a` и `b`.

Задача 2. Что будет напечатано в результате исполнения следующего фрагмента программы?

а)

```
struct st { char *s; } t1, t2;
t1.s = malloc(10);
strcpy (t1.s, "Hello ");
t2 = t1;
printf ("%s", t2.s);
strcpy (t1.s, "world!");
printf ("%s", t2.s);
```

Поскольку при копировании структур происходит побайтовое копирование значений их полей, то в результате выполнения присваивания `t2=t1` скопируются указатели: поле `t2.s` будет указывать на ту же область памяти, что и `t1.s` (сама область памяти скопирована не будет). В результате выведется: “Hello world!”.

б)

```
struct st { char s[10]; } t1, t2;
strcpy (t1.s, "Hello ");
t2 = t1;
printf ("%s", t2.s);
strcpy (t1.s, "world!");
printf ("%s", t2.s);
```

В данном случае полем структуры является массив из 10 символов, и при присваивании структур все его содержимое будет скопировано. В результате выведется “Hello Hello ”.

8.2. Объединения

Объединения в Си описываются так же, как и структуры, но для их описания используется ключевое слово `union`:

```
union u {
   int i;
   double f;
} tmp;
```

Объединения имеют одно принципиальное отличие от структур: все поля объединения располагаются в памяти начиная с одного и того же базового адреса и, таким образом, изменение одного поля объединения влечет за собой потенциальное изменение всех остальных полей объединения (в структуре же все поля независимы и располагаются в памяти без наложений друг на друга).

Обычно только одно из полей объединения имеет смысл в каждый конкретный момент времени. Какое из полей использовать – определяет программист. Например, описанную выше переменную `tmp` можно использовать для хранения целого числа или вещественного числа, но не для обоих чисел одновременно.

Объединения могут также использоваться для организации доступа к отдельным частям значений, например:

```
union intbytes {
   int number;
   unsigned char bytes[4];
} d;
d.number = 12345678;
printf ("Побайтовое представление числа %d в памяти"
         "имеет вид: %d %d %d %d\n",  d.number,
         d.bytes[0], d.bytes[1], d.bytes[2], d.bytes[3]);
```

Над объединениями определены те же операции, что и над структурами.

Размер объединения в памяти определяется размером наибольшего поля в объединении.

8.3. Задачи для самостоятельного решения

8.3.1. Описать заданное понятие в виде структуры. Описать также переменную этого типа и присвоить ей указанное значение:

а) дата (число, месяц, год); 16 ноября 1999 года;

б) адрес (страна, город, улица, дом, квартира); Россия, Москва, Ильинка, дом 3, кв. 34;

в) треугольник (две стороны и угол между ними); 5, 6.7, 35°;

8.3.2. Пусть «целочисленная» окружность на плоскости описана следующим образом:

```
struct point { int x; int y; };
struct circle { int radius; struct point center; };
```

Пусть есть массив, содержащий информацию об окружностях на плоскости:

```
struct circle plane[30];
```

Описать функцию, определяющую:

а) есть ли среди этих окружностей хотя бы две концентрические окружности;

б) есть ли среди этих окружностей хотя бы две вложенные (не обязательно концентрические) окружности;

в) есть ли среди этих окружностей хотя бы одна «уединенная», то есть не имеющая общих точек ни с какой другой окружностью массива plane.

8.3.3. Описать структурный тип для представления даты (день, месяц, год). Используя этот тип, описать функцию, принимающую на вход массив дат и упорядочивающую его по неубыванию.

8.3.4. Описать структуру для представления информации о человеке: фамилия (не более 30 символов), имя (не более 30 символов), возраст. Описать функцию, которая для заданного массива из описанных структур определяет:

а) возраст самого старшего человека;

б) количество людей с заданным именем (имя также является параметром функции);

в) количество людей, у которых есть однофамильцы;

г) фамилию человека, чей возраст ближе всего к среднему возрасту всех людей из массива (можно считать, что такой человек один).

8.3.5. Даны описания:

```
enum { circle = 0, triangle = 1, rectangle = 2};
// тип фигуры на плоскости

struct circle_info { double r; };
// информация об окружности: r – радиус

struct triangle_info { double a, b, c; };
// треугольник со сторонами a, b, c

struct rectangle_info { double w, h; };
/* прямоугольник: w – ширина,
                                        h – высота */
struct shape {
// описание одной фигуры на плоскости
     int kind; // 0, 1 или 2
     union {
          struct circle_info ci;
          struct triangle_info ti;
          struct rectangle_info ri;
     } info;
  };

struct shape plane[50];
// массив, описывающий все фигуры на плоскости
```

Описать функцию, которая для массива `plane` вычисляет:

а) количество окружностей на плоскости;

б) суммарный периметр всех фигур;

в) наибольшую площадь одной фигуры.

8.3.6. Используя описания из предыдущей задачи, описать функцию, которая заполняет массив plane следующим образом: первые 10 элементов – окружности с целочисленными радиусами от 1 до 10 (радиус окружности равен ее порядковому номеру); следующие 20 элементов – равнобедренные треугольники, основания

которых – целые числа от 1 до 20, а боковые стороны вдвое больше основания; последние 20 элементов – квадраты, стороны которых – целые числа от 1 до 20.

9. Файлы

С точки зрения программы файл представляет собой произвольную последовательность данных. Содержимое файла может быть интерпретировано как последовательность символов (текстовые файлы) или как двоичные данные (бинарные файлы).

Типы и функции для работы с файлами описаны в заголовочном файле `stdio.h`.

9.1. Открытие и закрытие файла

Для открытия файла используется функция *fopen*:

```
FILE *fopen(const char *filename, const char *mode);
```

Эта функция инициализирует работу с файлом и создает структуру со служебной информацией типа `FILE`. Точный состав этой структуры может зависеть от операционной системы. Функция возвращает указатель на эту структуру, который используется в дальнейшем для всех операций с файлом.

Функции передаются два параметра: параметр `filename` – имя открываемого файла, параметр `mode` – строка, определяющая режим работы с файлом. Допустимые значения режимов:

- `"r"` – существующий текстовый файл открывается для чтения;
- `"w"` – создается новый текстовый файл и открывается на запись. Если файл с таким именем существовал ранее, то его содержимое удаляется;
- `"a"` – текстовый файл открывается для записи в конец файла (его "старое" содержимое сохраняется) или создается;
- `"r+"` – существующий текстовый файл открывается для чтения и записи, текущая позиция (место в файле, по которому происходит чтение или запись) устанавливается в начало файла;
- `"w+"` – текстовый файл открывается или создается для чтения и записи, текущая позиция устанавливается в начало файла. Если файл с таким именем существовал ранее, то его содержимое удаляется;
- `"a+"` – текстовый файл открывается для чтения и записи, текущая позиция устанавливается в конец файла. Если файл не существовал, то он создается.

Для открытия файла в бинарном режиме к значению параметра `mode` приписывается буква `b`, например, `"rb"` означает открытие существующего бинарного файла на чтение, а `"a+b"` (или `"ab+"`) – открытие бинарного файла для чтения и записи с позиционированием в конец файла.

Отличие текстового режима от бинарного в том, что в текстовом режиме некоторые последовательности символов могут интерпретироваться особым образом. Набор специальных последовательностей и их интерпретация зависят от операционной системы.

Так, например, на операционной системе Windows перевод строки в текстовых файлах записывается как последовательность из двух символов: символ с кодом 13 и символ с кодом 10. В случае открытия файла как текстового, данная последовательность будет проинтерпретирована как один символ '\n', и именно в таком виде будет прочитана функциями стандартной библиотеки. В случае же открытия файла как бинарного, она будет воспринята как последовательность из двух байт со значениями 13 и 10.

В случае успешного открытия файла функция fopen возвращает указатель на переменную типа FILE. В случае ошибки функция возвращает значение NULL.

После окончания работы с файлом необходимо убедиться, что все записанные данные попали на диск, и освободить все ресурсы, связанные с ним. Для этого используется функция закрытия файла fclose:

```
int fclose(FILE *stream);
```

Функция производит дозапись еще не записанных данных в файл и освобождает все ресурсы, связанные с открытым файлом. В случае успешного закрытия файла функция возвращает нулевое значение, в случае ошибки – значение EOF (константа EOF также описана в stdio.h и является отрицательным целым числом).

9.2. Стандартные потоки ввода-вывода

Стандартный поток ввода (обычно – ввод с клавиатуры) и стандартный поток вывода (обычно – вывод на экран) с точки зрения программы также являются файлами. Для доступа к ним можно использовать описанные в stdio.h переменные stdin и stdout соответственно; например, вызов printf ("Hello\n") эквивалентен вызову fprintf (stdout, "Hello\n"). В отличие от обычных файлов, к стандартным потокам нельзя применять операции позиционирования, а только последовательного чтения или записи.

Третьим стандартным потоком является stderr – стандартный поток для сообщений об ошибках и диагностических сообщений. Стандартный поток stderr также обычно ассоциируется с экраном и не буферизуется, сообщения, выведенные в него, сразу появляются на экране. Отдельный поток для сообщений об ошибках выделяется для того, чтобы можно было разделить нормальный вывод программы и диагностические сообщения. Вывод сообщения в поток stderr производится с помощью вызова функции записи в файл с указанием stderr как имени файла:

```
fprintf (stderr, "Ошибка выделения памяти"\n);.
```

9.3. Чтение из файла и запись в файл

Чтение и запись выполняются в текущей позиции файла. После чтения (или записи) текущая позиция сдвигается к концу файла на количество считанных (или записанных) символов. Чтение и запись являются *буферизованными*, т.е., например, окончание работы функции записи не означает немедленной модификации содержимого файла на диске: результаты записи могут быть отражены во внутреннем буфере

библиотеки работы с файлами, которые будут сброшены при закрытии файла или при вызове функции fflush.

Для функций форматного ввода-вывода printf и scanf существуют аналоги при работе с файлами: fprintf и fscanf:

```
int fprintf(FILE *stream, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
```

Использование этих функций – такое же, как и функций printf и scanf, но в качестве первого параметра указывается файл, с которым ведется работа. Функция fprintf возвращает количество записанных символов, функция fscanf – количество успешно прочитанных значений (или EOF в случае конца файла или ошибки чтения).

Посимвольное чтение осуществляется функцией fgetc:

```
int fgetc(FILE *stream);
```

Функция возвращает очередной символ из файла в виде unsigned char, преобразованного к int, или значение EOF, если файл исчерпан или обнаружена ошибка.

Для посимвольного вывода служит функция fputc:

```
int fputc(int c, FILE *stream);
```

Функция записывает символ *c*, преобразованный к типу unsigned char, в указанный файл и возвращает значение записанного символа (или значение EOF в случае ошибки).

Чтение из файла строки (под строкой файла понимают последовательность символов, оканчивающуюся символом конца строки, '\n') осуществляется функцией fgets:

```
char *fgets(char *s, int size, FILE *stream);
```

Функция читает в массив s не более (size-1) символов, прекращая чтение, если встретился символ конца строки (он при этом также заносится в s) или конец файла. За прочитанными символами в массиве записывается символ '\0'. Функция возвращает s в случае успешного завершения, или NULL – в случае ошибки, а также если файл оказался исчерпан до того, как был прочитан хотя бы один символ.

Функция fgets не имеет своего аналога для работы со стандартными потоками ввода-вывода, так как аналогичная функция gets не принимает параметра size и продолжает чтение до тех пор, пока есть данные, невзирая на границы массива, в который производится чтение. Поэтому организовать безопасное чтение строк с проверкой их размера невозможно с помощью функции gets, и необходимо использовать функцию fgets из файла sdtin вместо нее.

Запись строки в файл выполняется при помощи функции fputs:

```
int fputs(const char *s, FILE *stream);
```

Функция записывает в файл все символы строки *s* (терминальный '\0' в файл не пишется). В случае ошибки функция возвращает EOF, в противном случае – неотрицательное число. Действие функции эквивалентно вызову:

```
fprintf (stream, "%s", s);
```

Пример 1. Дан текстовый файл input.txt, содержащий целые числа. Требуется напечатать сумму всех чисел в нем.

```
FILE *f;
int sum = 0, n;
f = fopen("input.txt", "r");
while (fscanf (f, "%d", &n) == 1)
  sum += n;
fclose (f);
printf (”%d\n”, sum);
```

9.4. Бинарные файлы

При работе с бинарными данными в файле функции `fgets/fputs`, `fscanf/fprintf` использовать невозможно, поскольку они интерпретируют содержимое файла как текст — то есть последовательность строковых данных. Двоичные же данные записываются ровно в том виде, в котором они представляются в памяти. Например, на архитектуре little-endian с 32-битным целым типом запись одной переменной целого типа в файл должна поместить в файл 4 байта, находящиеся в памяти по адресу, где хранится переменная: первым запишется младший (нулевой) байт, последним — старший байт. Аналогично, при чтении последовательные байты файла инициализируют последовательные байты в памяти, поэтому на одной и той же архитектуре считанный объект некоторого типа будет равен предварительно записанному в это место файла объекту того же типа.

Если в файле записана последовательность байт с кодами `0x31 0x32 0x33 0x34`, то как текстовый в кодировке ASCII он будет считан как строка `"1234"`. Функция `fscanf` с форматным преобразованием `"%d"` считает число 1234. В двоичном же виде должно быть считано число $\texttt{0x34333231}_{16} = 875770417_{10}$.

Для чтения и записи в/из файлов данных в двоичном виде используются функции `fread` и `fwrite`.

Функция `fwrite` объявлена в стандартной библиотеке Си как

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *fp);
```

Функция записывает в файл, на который указывает дескриптор `fp`, `nmemb` объектов, каждый из которых имеет размер `size`, и берет эти объекты последовательно по адресу `ptr`. Можно представлять себе, что `ptr` должен указывать на массив из `nmemb` элементов, в котором хранятся объекты размера `size`. Функция возвращает количество успешно записанных элементов, что обычно совпадает с `nmemb`.

Для записи массива из 10 целых элементов:

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
FILE *f = fopen ("out.bin", "wb");
fwrite (a, sizeof (int), 10, f);
```

или:

```
fwrite (a, sizeof (a), 1, f);
```

Для записи одного числа `double` или структуры:

```
struct point { int x, y; };
struct point pt = {3, 3}, *p = &pt; double d;
fwrite (p, sizeof (struct point), 1, f);
fwrite (&d, sizeof (d), 1, f);
```

Обратите внимание, что из-за необходимости передавать адреса любых объектов первый параметр `fwrite` имеет тип `const void *`.

Функция `fread` имеет тот же прототип, но на этот раз первый аргумент `ptr` указывает на область памяти, в которую будут последовательно считаны `nmemb` объектов, каждый из которых имеет размер `size`. Аналогично, возвращается количество успешно считанных объектов. Если был достигнут конец файла или произошла другая ошибка, то возвращаемое значение будет меньше `nmemb`. Используйте этот факт для проверки корректности ввода.

Для считывания назад записанного массива `a` или числа `d`:

```
FILE *f = fopen ("out.bin", "rb");
fread (a, sizeof (int), 10, f);
fread (&d, sizeof (double), 1, f);
```

9.5. Позиционирование в файле

Функция `fseek` устанавливает текущую позицию в файле:

```
int fseek(FILE *stream, long offset, int origin);
```

Новое значение текущей позиции, измеряемое в символах (байтах), получается добавлением `offset` байтов к позиции, указанной параметром `origin`. Возможные значения `origin`:

- `SEEK_SET` – смещение отсчитывается относительно начала файла (параметр `offset` должен быть неотрицателен);
- `SEEK_CUR` – смещение отсчитывается относительно текущей позиции в файле (параметр `offset` может иметь любой знак);
- `SEEK_END` – смещение отсчитывается относительно конца файла (`offset` должен быть меньше либо равен нуля).

Константы `SEEK_SET`, `SEEK_CUR`, `SEEK_END` также описаны в `stdio.h`. В случае успешного завершения `fseek` возвращает 0, в противном случае – -1.

Узнать значение текущей позиции в файле можно при помощи функции `ftell`:

```
long ftell(FILE *stream);
```

Функция возвращает значение текущей позиции (смещение в байтах относительно начала файла) или -1 в случае ошибки.

Пример 2. Определить и напечатать размер файла `input.txt`.

```
FILE *f;
long size;
f = fopen ("input.txt", "r");
if (f != NULL) {
```

```
  fseek (f, 0, SEEK_END);
  size = ftell (f);
  fclose (f);
  printf ("Размер файла - %ld байт.\n", size);
} else {
  printf ("Не удалось открыть файл input.txt\n");
}
```

Задача 1. Файл `a.txt` содержит два целых числа. Требуется дописать их сумму в конец этого же файла.

```
#include <stdio.h>
int main (void) {
  FILE *f;
  int a, b;
  f = fopen ("a.txt", "r+");
  fscanf (f, "%d%d", &a, &b);
  fprintf (" %d", a+b);
  fclose (f);
  return 0;
}
```

Задача 2. Требовалось описать функцию, которая для заданного файла, уже открытого на чтение, подсчитывает количество десятичных цифр в нем. Какие из приведенных ниже описаний правильно решают эту задачу?

а)

```
int digits (FILE *f) {
  unsigned char c;
  int result = 0;
  while ((c = fgetc (f)) != EOF) {
    if (c >= '0' && c <= '9') result++;
  }
  return result;
}
```

Цикл `while` в данной функции никогда не завершится: при достижении конца файла значение константы `EOF` будет приведено к типу `unsigned char` в момент присваивания `c = fgetc (f)`, и поскольку эта константа отрицательна, то такое после приведения ее знак изменится, и последующее сравнение с исходным значением `EOF` будет ложно.

б)

```
int digits (FILE *f) {
  signed char c;
  int result = 0;
  while ((c = fgetc (f)) != EOF) {
    if (c >= '0' && c <= '9') result++;
  }
  return result;
}
```

Описание этой функции также ошибочно. Константа `EOF` не принадлежит типу `char` (ее тип – `int`). В большинстве реализаций значение `EOF` равно -1, и цикл успешно

завершится по достижении конца файла. Однако он завершится и при чтении символа, код которого при приведении к типу `signed char` окажется равным -1. Например, при обработке файла в кодировке Windows-1251 первая же буква "я" (имеющая код 255) завершит работу такого цикла (поскольку (`signed char`) 255 = -1).

в)

```
int digits (FILE *f) {
  int c;
  int result = 0;
  while ((c = fgetc (f)) != EOF) {
    if (c >= '0' && c <= '9') result++;
  }
  return result;
}
```

Задача решена верно.

9.6. Задачи для самостоятельного решения

В задачах 9.6.1.— 9.6.9. требуется написать полную программу, обрабатывающую текстовый файл с именем `input.txt`.

9.6.1. Определить, сколько раз в файле встречается последовательность символов for.

9.6.2. Распечатать все строки файла, содержащие заданную строку (вводится с клавиатуры) в качестве подстроки. Известно, что длина строки в файле не превосходит 80 символов.

9.6.3. Определить, какая строка является самой длинной в файле. Если таких строк несколько, то выдать первую из них.

9.6.4. В файле записана непустая последовательность целых чисел. Требуется:

а) найти наибольшее из этих чисел;

б) подсчитать количество четных чисел;

в) определить, составляют ли эти числа арифметическую прогрессию;

г) определить, составляют ли эти числа геометрическую прогрессию;

д) определить, сколько чисел этой последовательности являются точными квадратами.

9.6.5. Дописать в конец файла строку `FINISH`.

9.6.6. Создать файл с именем `copy.txt` – копию заданного файла.

9.6.7. Удалить из файла все пустые строки.

9.6.8. В файле записана непустая последовательность целых чисел, являющихся числами Фибоначчи. Приписать еще одно, очередное число Фибоначчи.

9.6.9. Изменить файл следующим образом: перед каждой строкой добавить ее номер и пробел. Длина строки в исходном файле не превосходит 80 символов.

9.6.10. Файлы `a.txt` и `b.txt` содержат последовательности целых чисел (возможно, пустые), причем числа в каждом файле упорядочены по неубыванию. Написать программу, которая создает файл `c.txt`, содержащий все числа из обоих

входных файлов и также упорядоченный по неубыванию. Запрещается использовать массивы и динамическую память.

10. Динамические структуры данных

Динамические структуры данных используются для представления в памяти данных, размер которых заранее неизвестен. В дальнейшем будут рассматриваться динамические структуры данных, состоящие из однотипных элементов (звеньев), некоторые из которых содержат ссылки на другие звенья.

При реализации таких конструкций на языке Си для представления звеньев используются структуры (`struct`), а для представления ссылок – указатели. Память под звенья выделяется динамически (например, функцией `malloc`) и должна освобождаться (функцией `free`) при удалении звеньев.

10.1. Списки

Линейный однонаправленный список – структура данных, в которой каждое звено содержит ссылку на следующее звено. Например, можно привести такое описание списка, каждое звено которого хранит целое число в качестве данных:

```
struct listnode {
  int elem; // хранимое значение
  struct listnode *next; // указатель на следующее звено
};
```

Последнее звено списка содержит в поле `next` значение `NULL` (для некольцевых списков) или указатель на первое звено списка (для кольцевых списков).

В звеньях *двунаправленного списка* хранятся ссылки не только на следующее звено, но и на предыдущее:

```
struct listnode {
  int elem; // хранимое значение
  struct listnode *prev, *next; // предыдущее и следующее звено
};
```

Для работы со списком достаточно указателя на его первое звено: все остальные звенья можно получить, последовательно просматривая список. Поэтому при передаче списка в качестве параметра передают только этот указатель. Для списка, не содержащего ни одного звена, значение такого указателя принимают равным `NULL`.

Пример 1. Описана переменная

```
struct listnode *L;
```

Построить список из трех звеньев, содержащий числа 1, 2, 3, и занести в `L` указатель на его начало.

```
struct listnode *p;

// создание первого звена списка
p = (listnode*) malloc (sizeof (struct listnode));
p->elem = 1;
```

```
// создание второго звена списка
p->next = (listnode*) malloc (sizeof (struct listnode));
p->next->elem = 2;
L = p;

// сдвиг p на второе звено списка
p = p->next;

// создание последнего звена списка
p->next = (listnode*) malloc (sizeof (struct listnode));
p->next->elem = 3;
p->next->next = NULL; // следующего звена нет
```

Пример 2. Найти сумму всех элементов списка L.

```
struct listnode *p;
int sum = 0;
p = L;
while (p != NULL) {
  sum += p->elem;
  p = p->next; // движение по списку вправо
}
printf ("%d\n", sum);
```

Для облегчения операций добавления и удаления звеньев в списки используют списки с заглавным звеном. В таких списках всегда содержится хотя бы одно (первое) звено, которое и называют заглавным; в заглавном звене не хранят никаких данных. Для списка с заглавным звеном указатель на его начало никогда не изменяется, поскольку заглавное звено списка не меняется. Это позволяет писать более компактный код.

Пример 3. Описать функцию, которая добавляет в конец заданного списка звено с заданным целым значением.

а) для списка без заглавного звена:

```
void insert (struct listnode **L, int x) {
  struct listnode *p, *q;
  p = (struct listnode*) malloc (sizeof (struct listnode));
  p->elem = x;
  p->next = NULL;
  if (*L == NULL) {
    // пустой список: меняется указатель на начало
    *L = p;
  } else {
    q = *L;
    // непустой список: находим последнее звено
    while (q -> next != NULL)
      q = q->next;
    q->next = p;
  }
}
```

Пример вызова такой функции:

```
insert (&L, 10);
```

Другой вариант решения этой задачи – возвращать указатель на новое начало списка, чтобы избежать работы с двойными указателями:

```
struct listnode * insert (struct listnode *L, int x) {
  struct listnode *p, *q;
  p = (struct listnode*) malloc (sizeof (struct listnode));
  p->elem = x;
  p->next = NULL;
  if (L == NULL) {
    // список был пуст, новый список будет
    return p;  // начинаться с указателя p
  } else {
    q = L; // находим последнее звено
    while (q->next != NULL)
      q = q->next;
    q->next = p;
    return L; // указатель на начало списка не изменился
  }
}
```

Пример вызова такой функции:

```
L = insert (L, 10);
```

б) для списка с заглавным звеном:

```
void insert (struct listnode *L, int x) {
  struct listnode *p, *q;
  p = (struct listnode*) malloc (sizeof (struct listnode));
  p->elem = x;
  p->next = NULL;
  q = L; // список точно не пуст: есть хотя бы заглавное звено
  while (q -> next != NULL)
    q = q->next;
  q->next = p;
}
```

Пример вызова такой функции:

```
insert (L, 10);
```

В дальнейшем будут подразумеваться однонаправленные некольцевые списки без заглавного звена, если иное не указано явно.

Пример 4. Описать функцию, удаляющую все звенья из заданного списка.

```
struct listnode* erase (struct listnode *L) {
  struct listnode *p;
  while (L != NULL) {
    p = L;
    L = L->next;
    free (p);
  }
  return NULL;
}
```

Рекурсивное решение нередко оказывается короче нерекурсивного (хотя быстродействие рекурсивных решений, как правило, ниже):

```
struct listnode *erase (struct listnode *L) {
  if (L != NULL) {
    erase (L->next);
    free (L);
  }
  return NULL;
}
```

10.1.1. Задачи для самостоятельного решения

Списки.

Замечание. В задачах 10.1.1. – 10.1.11. считать, что под списком подразумевается однонаправленный некольцевой список без заглавного звена, в звеньях которого хранятся целые числа `(int)`.

10.1.1. Описать функцию `sum2(L)`, возвращающую сумму последнего и предпоследнего чисел в списке `L` (если они есть).

10.1.2. Описать функцию `max(L)`, возвращающую максимальное значение в непустом списке L.

10.1.3. Описать функцию has_`duplicate(L)`, определяющую, есть ли в списке `L` хотя бы два звена с одинаковыми числами.

10.1.4. Описать функцию, которая получает в качестве параметра массив и строит по нему список, содержащий те же элементы в том же порядке. В качестве результата функция возвращает указатель на начало построенного списка. (Рекомендация: строить список с конца.)

10.1.5. Описать функцию `copy(L)`, которая строит копию списка `L` и возвращает указатель на начало нового списка.

10.1.6. Описать функцию `reverse(L)`, изменяющую порядок чисел в списке `L` на противоположный. (Рекомендация: строить новый список, добавляя элементы в его начало.)

10.1.7. Описать функцию `insert(L, X)`, которая вставляет в упорядоченный по неубыванию список `L` число `X` так, чтобы результирующий список был также упорядочен по неубыванию.

10.1.8. Описать функцию `same(L1, L2)`, которая определяет, являются ли списки `L1` и `L2` одинаковыми (назовем списки одинаковыми, если они содержат одни и те же числа в одинаковом порядке).

10.1.9. Описать функцию `filter(L1, L2)`, которая удаляет из списка `L1` все числа, которые содержатся в списке `L2`.

10.1.10. Описать функцию `sort(L)`, упорядочивающую список `L` по неубыванию.

10.1.11. Описать функцию `merge(L1, L2)`, которая из двух упорядоченных по неубыванию списков `L1` и `L2` строит новый список, содержащий все числа из этих двух списков и также упорядоченный по неубыванию.

а) новый список строится из звеньев `L1` и `L2`; функция не должна создавать новых звеньев;

б) новый список строится из копий звеньев L1 и L2; списки L1 и L2 не изменяются.

10.1.12. Считая, что L – однонаправленный кольцевой список без заглавного звена, описать функцию insert(L, X), которая вставляет число X в начало списка L.

10.1.13. Считая, что L – двунаправленный некольцевой список без заглавного звена, описать функцию delete(L, X), которая удаляет из списка L все звенья, хранящие число X (если они есть).

10.2. Стек. Очередь

На практике нередко встречается необходимость откладывать обработку некоторых данных на более позднее время. Для организации такой отложенной обработки используют, в частности, стеки и очереди. И стек, и очередь определяют две основные операции: добавление элемента в структуру данных и извлечение первого элемента из структуры данных.

Стек (англ. stack) – структура данных, предоставляющая доступ к элементам в порядке, обратном порядку добавления. То есть элемент, добавленный в стек последним, будет возвращен первой же операцией извлечения. Операцию добавления в стек принято называть push, а операцию извлечения из стека – pop.

В случае *очереди* элементы извлекаются в том же порядке, в котором они были добавлены. Функции, реализующие добавление и извлечение из очереди, рекомендуется называть enqueue и dequeue соответственно.

Стек и очередь могут быть реализованы, например, с помощью однонаправленных списков или с помощью динамических массивов; выбор зависит от конкретной задачи. Всю работу по поддержанию стека или очереди обычно выносят в отдельный модуль (достаточно описать пять функций: создание структуры данных, ее уничтожение, добавление нового элемента, извлечение элемента, проверка на пустоту). Такой подход позволяет отделить логику решения задачи от реализации структуры данных, благодаря чему можно, например, легко заменить в программе одну реализацию стека на другую или использовать одну и ту же реализацию стека в разных задачах.

Пример реализации стека с помощью динамического массива:

```
typedef int datatype; // тип данных, хранимых в стеке
typedef struct { //  структура - представление стека
  datatype *items;
  int size; // размер стека
  int sp; // номер последнего занятого
 } stack;

  // функции для работы со стеком
  void init_stack (stack *st) {
    st->size = 16;
    st->sp = -1;
    st->items = malloc (16 * sizeof (datatype));
  }
```

```
void delete_stack (stack *st) {
  free (st->items);
}
void push (stack *st, datatype value) {
  if (st->sp == st->size - 1) {
    st->size = st->size * 2;
    st->items = realloc (st->items,
                         st->size * sizeof (datatype));
  }
  st->items[++st->sp] = value;
}
void pop (stack *st, datatype *value) {
  if (st->sp < 0) {
    printf ("Стек пуст!");
    exit (1);
  }
  *value = st->items[st->sp--];
}
int empty_stack (stack *st) {
  return st->sp < 0;
}
```

Пример использования стека.

Считая, что описан тип `stack (datatype = char)`, а также функции `init_stack`, `delete_stack`, `push`, `pop`, `empty_stack` для работы со стеком, описать нерекурсивную функцию `reverse`, которая вводит с клавиатуры текст — последовательность символов, оканчивающуюся точкой, и выводит эту последовательность в обратном порядке.

```
void reverse (void) {
  stack st;
  char c;
  init_stack (&st);
  while ((c = getchar ()) != '.')
    push (&st, c);
  while (!empty_stack (&st)) {
    pop (&st, &c);
    putchar (c);
  }
  delete_stack (&st);
}
```

Пример реализации очереди с помощью массива:

```
typedef int datatype; // тип данных, хранимых в очереди
enum { max_queue_size = 256 };
typedef struct { // структура - представление очереди
  datatype items[max_queue_size];
  int head; // индекс первого элемента в очереди
  int size; // количество элементов в очереди
} queue;

  // функции для работы с очередью
```

```
void init_queue (queue *q) { q->head = 0; q->size = 0; }
void delete_queue (queue *q) { }
void enqueue (queue *q, datatype value) {
  if (q->size >= max_queue_size) {
    printf ("Очередь переполнена!\n");
    exit (1);
  }
  q->items[(q->head + q->size) % max_queue_size] = value;
  q->size++;
}
void dequeue (queue *q, datatype *value) {
  if (q->size == 0) {
    printf ("Очередь пуста!\n");
    exit(1);
  }
  *value = q->items[q->head];
  q->head = (q->head + 1) % max_queue_size;
  q->size--;
}
int empty_queue (queue *q) { return q->size == 0; }
```

10.2.1. Задачи для самостоятельного решения

Стек. Очередь.

10.2.1. Описать реализацию стека на основе однонаправленного списка.

10.2.2. Описать реализацию очереди на основе однонаправленного списка.

В задачах 10.2.3. — 10.2.6. считать уже описанными типы и функции для работы со стеком и очередью.

10.2.3. Написать программу, которая переводит заданное алгебраическое выражение, содержащее операнды (целые числа), круглые скобки, бинарные операции + и −, в постфиксную запись. Считать точку признаком конца выражения. Для решения задачи использовать стек.

10.2.4. Решить предыдущую задачу, если в качестве операций могут встречаться также умножение и деление (* и /); приоритеты операций должны быть соблюдены.

10.2.5. Реализовать вычисление выражения, содержащего бинарные операции +, −, *, / и записанного в постфиксной форме. В реализации использовать стек.

10.2.6. Дан файл, содержащий целые числа (их количество заранее не известно). Определить подходящую структуру данных и написать программу, которая за один просмотр файла решает следующую задачу:

а) напечатать сумму первого и последнего чисел, сумму второго и предпоследнего чисел и т. д.;

б) вывести сначала все четные числа файла, а затем — все нечетные в том же порядке, в котором они идут в файле;

в) вывести все четные числа из файла в исходном порядке, а затем — все нечетные числа файла в порядке, обратном порядку их следования в файле.

10.3. Двоичные деревья

Деревья используются преимущественно для хранения упорядоченных данных с возможностью быстрого поиска по ключу.

Двоичное дерево — древовидная структура данных, в которой каждая вершина имеет не более двух потомков. Можно привести рекурсивное определение двоичного дерева:

a) пустая структура (пустое множество вершин) является двоичным деревом;

b) структура, состоящая из вершины, потомками которой являются два двоичных дерева (возможно, одно из них или оба — пустые), также образует двоичное дерево.

Любое поддерево двоичного дерева также является деревом.

Корнем дерева называется вершина, не являющаяся потомком никакой другой вершины. В непустом дереве всегда ровно один корень.

Листьями дерева называются вершины, не имеющие потомков.

Высотой дерева называется наибольшая длина пути от корня дерева до любого из его листьев. Длина пути измеряется в пройденных вершинах: так, у дерева, состоящего из единственной вершины, корень является и его единственным листом, а высота такого дерева равна 1.

Легко видеть, что количество листьев в двоичном дереве высоты `h` не превышает 2^{h-1}, а общее количество вершин ограничено величиной 2^h-1.

Для работы с двоичными деревьями на языке Си можно использовать следующее описание:

```
struct treenode {
  int elem;
  struct treenode *left, *right;
};
```

При передаче двоичного дерева в функцию передают указатель на корень этого дерева. Пустому дереву соответствует указатель `NULL`.

Для обхода всех вершин дерева используют как рекурсивные, так и нерекурсивные алгоритмы.

Рекурсивный обход дерева вытекает непосредственно из рекурсивного определения дерева. Пример: функция, печатающая значения всех элементов дерева.

```
void print_tree (struct treenode *T) {
  if (T != NULL) {
    printf ("%d\n", T->elem);
    print_tree (T->left);
    print_tree (T->right);
  }
}
```

Рекурсивный алгоритм осуществляет обход дерева «в глубину»: элементы дерева, расположенные дальше от корня, обрабатываются раньше большинства (или всех) элементов, находящихся ближе к корню.

При нерекурсивном обходе дерева для хранения вершин, подлежащих обработке, используют стек или очередь. Использование стека дает тот же порядок обработки вершин, что и приведенный выше рекурсивный алгоритм, а использование очереди

позволяет обрабатывать элементы дерева в порядке увеличения расстояния от корня: сначала корневую вершину, затем – все вершины, отстоящие от корня на 1, затем – все вершины, отстоящие от корня на 2, и так далее. Такой обход называется обходом «в ширину».

Пример: нерекурсивные функции, печатающие значения всех элементов дерева при помощи стека и очереди (стек и очередь при этом должны хранить значения типа `struct treenode *`).

```
// typedef struct treenode *datatype;
// тип данных для стека/очереди
    void print_tree_stack (struct treenode *T) {
      stack st;
      init_stack (&st);
      push (&st, T);
      while (!empty_stack (&st)) {
        pop (&st, &T);
        if (T != NULL) {
          printf ("%d\n", T->elem);
          push (&st, T->right);
          push (&st, T->left);
        }
      }
      delete_stack (&st);
    }

    void print_tree_queue( struct treenode *T) {
      queue q;
      init_queue (&q);
      enqueue (&q, T);
      while (!empty_queue (&q)) {
        dequeue (&q, &T);
        if (T == NULL)
          continue;
        printf ("%d\n", T->elem);
        enqueue (&q, T->left);
        enqueue (&q, T->right);
      }
      delete_queue (&q);
    }
```

10.3.1. Задачи для самостоятельного решения

10.3.1. Описать функцию, которая:

а) вычисляет сумму всех элементов двоичного дерева;

б) подсчитывает количество вхождений заданного элемента в двоичное дерево;

в) вычисляет высоту двоичного дерева;

г) печатает значения данных из всех вершин дерева, не являющихся листьями;

д) проверяет, идентичны ли два двоичных дерева;

е) удаляет все элементы заданного двоичного дерева.

10.3.2. Используя определенные ранее типы данных «очередь» и «стек», описать нерекурсивную функцию, которая:

а) определяет число вхождений заданного элемента в двоичное дерево;

б) вычисляет сумму элементов двоичного дерева;

в) находит длину (количество узлов) на пути от корня до ближайшей вершины, содержащей заданный элемент (если такой вершины в дереве нет, то считать результат равным -1);

г) подсчитывает количество узлов на заданном уровне непустого двоичного дерева (корень считать вершиной нулевого уровня).

10.4. Деревья поиска. АВЛ-деревья

Двоичным деревом поиска называется двоичное дерево, в котором левое поддерево каждой вершины может содержать только значения (ключи), строго меньшие ключа в вершине, а правое поддерево – только ключи, строго большие ключа в вершине. Одинаковых ключей в дереве поиска быть не может.

Дерево поиска позволяет находить нужные элементы в нем за время, пропорциональное высоте дерева. Пример: функция, возвращающая указатель на вершину дерева поиска с заданным ключом.

```
struct treenode *search (struct treenode *T, int key) {
  if (T == NULL)
    return NULL;
  if (T->elem == key)
    return T;
  if (key < T->elem)
    return search (T->left, key);
  else
    return search (T->right, key);
}
```

Эффективность поиска ключа в дереве поиска существенно зависит от его структуры. Так, для дерева поиска из N вершин в лучшем случае его высота составляет приблизительно $\log_2 N$ (и при поиске в дереве из миллиона вершин придется выполнить всего 20 сравнений), а в худшем – высота равна N: дерево «вытянуто» в цепочку (и при поиске в таком дереве придется просмотреть все его вершины).

АВЛ-дерево – двоичное дерево поиска, в каждой вершине которого высота левого поддерева этой вершины отличается от высоты правого поддерева этой вершины не более чем на единицу. Такое свойство существенно ограничивает увеличение высоты дерева с ростом количества вершин, то есть поиск нужного элемента происходит быстро. Кроме того, существует эффективный алгоритм добавления (вставки) нового элемента в АВЛ-дерево: новый элемент добавляется в качестве листа к одному из поддеревьев (как и в дереве поиска), а затем для тех вершин, где условие балансировки АВЛ-дерева оказалось нарушенным, выполняется соответствующий *поворот,* который сводится к перестановке нескольких вершин и поддеревьев. Пример: реализация короткого правого поворота RR.

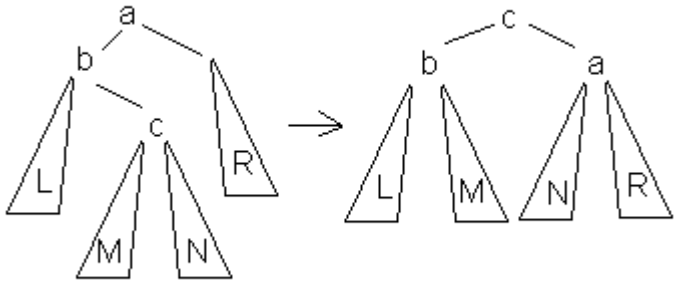

Рис. 6. Балансировка АВЛ-дерева

```
struct treenode *rotate_RR (struct treenode *root) {
  struct treenode *b = root->left;
  struct treenode *c = b->right;
  root->left = c->right; // перенос поддерева N
  c->right = root; // a становится потомком c
  b->right = c->left; // перенос поддерева M
  c->left = b; // b становится потомком c
  return c; // указатель на новый корень дерева
}
```

10.4.1. Задачи для самостоятельного решения

10.4.1. Описать функцию, которая определяет, является ли данное двоичное дерево деревом поиска:

а) рекурсивно;

б) нерекурсивно, используя определенные ранее типы данных «стек» и «очередь».

10.4.2. Описать функцию, которая для заданного двоичного дерева поиска:

а) находит наименьшее значение в нем (считать, что дерево непустое);

б) добавляет заданный элемент в дерево поиска (если элемент уже был в дереве, то не добавлять);

в) удаляет из непустого дерева поиска его максимальный элемент;

г) находит в дереве поиска максимальное и второе по величине значения (считать, что дерево содержит не менее двух элементов);

д) удаляет из дерева поиска элемент с заданным значением (если он есть);

е) определяет, является ли заданное дерево поиска АВЛ-деревом.

11. Схема компиляции Си-программы. Препроцессор. Компоновщик. Отладчик

Реальные программы на Си, как правило, состоят из нескольких файлов. Среди этих файлов могут быть заголовочные файлы, содержащие *интерфейс* программы или ее частей (т.е. набор функций и переменных, с помощью которых можно использовать функциональность, реализованную программой), и файлы с реализацией частей

программы (с расширением `.c`). В Си используется раздельная компиляция, то есть все файлы с кодом программы можно компилировать отдельно.

Схема компиляции Си-программы представлена на рисунке 7. В ходе компиляции одного файла сначала этот файл обрабатывается *препроцессором* – компонентом, производящим набор текстовых подстановок над файлом для получения его окончательного вида и передачи компилятору. Потом компилятор получает *ассемблерный* код, то есть представляет программу в виде последовательности команд центрального процессора и описания необходимых ячеек в глобальной и статической памяти. Далее *ассемблер* получает по файлу на языке ассемблера т.н. *объектный* файл (как правило, имеющий расширение `.o` на Unix-подобных системах), в котором команды ассемблера закодированы в двоичном виде, а также описан вид статической и глобальной памяти. Наконец, *компоновщик* обеспечивает слияние нескольких объектных файлов в один исполняемый файл программы.

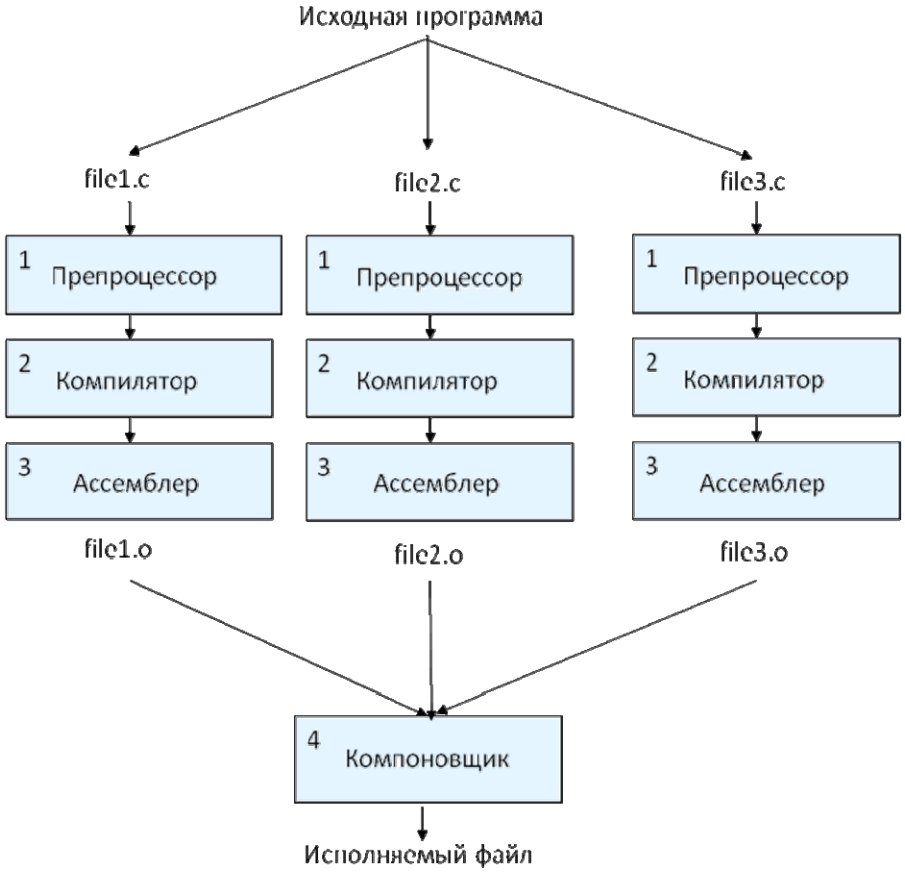

Рис. 7. Схема компиляции многомодульной программы.

11.1. Препроцессор

Препроцессор Си обеспечивает текстовую обработку файла программы до передачи ее компилятору. Среди задач препроцессора можно назвать обеспечение модульности программы (т.е. описание интерфейсных функций программы в заголовочных файлах и их подключение), обеспечение переносимости программы (за счет условной компиляции), автоматическая генерация однотипных участков кода (за счет макросов). Управление препроцессором обеспечивается включением в текст программы директив препроцессора. При обработке каждой директивы препроцессор меняет текст

программы определенным образом, после чего удаляет из текста обработанную директиву.

После окончания работы препроцессора компилятору Си на вход подается файл с текстом программы, содержащим лишь определения и объявления функций и объявления переменных, как описано в разделе 1. Строго говоря, до этапа препроцессирования компилятор также производит предварительную подготовку текста программы – обрабатывает многобайтовые символы, заменяет комментарии пробелом и т.п., – но рассмотрение этого процесса не входит в задачи настоящего пособия.

Подключение заголовочных файлов программы осуществляется с помощью директивы `#include <file.h>` или `#include "file.h"`. Как правило, список директив `#include` для подключения необходимых файлов указывается в первых строках .c-файла. Первый вариант директивы с заключением имени файла в угловые скобки используется для общесистемных библиотек, включая стандартную библиотеку Си, а второй вариант (с двойными кавычками) – для пользовательских файлов. В скобках или кавычках находится имя получаемого файла или относительный путь к нему. Разница в двух вариантах директивы заключается в разных наборах каталогов, относительно которых ищутся заданные в директиве пути к подключаемым заголовочным файлам.

Директива `#define` служит для определения некоторого имени, называемого *макросом*, и соответствующего ему *текста подстановки*, или *тела макроса*. Например, директива

```
#define DEBUGPRINT fprintf (stderr, "debug output:")
```

определяет макрос `DEBUGPRINT`, при этом после обработки директивы далее в тексте программы препроцессор заменяет все вхождения имени макроса в текст на его тело (выполняется т.н. *макроподстановка*). Так, строка `DEBUGPRINT;` будет заменена на строку `fprintf (stderr, "debug output:");`. Макроподстановка не будет выполнена только в том случае, если имя макроса будет найдено внутри строки, заключенной в двойные кавычки: строка `printf ("DEBUGPRINT: x = %d\n", x);` не будет изменена препроцессором.

На имя макроса налагаются те же ограничения, что и на имена переменных в Си – имя состоит из цифр, букв и знака подчеркивания (_), первый символ имени не может быть цифрой. Традиционно в именах макросов используются заглавные буквы. Макрос считается определенным от строки, содержащей соответствующую директиву `#define`, до конца программного файла. Если необходимо переопределить макрос или сделать его не определенным, то необходимо использовать директиву `#undef macro` для удаления определения макроса `macro`, после чего можно будет снова определить макрос с тем же именем, но другим телом.

Если в ходе макроподстановки получившаяся строка снова содержит известные препроцессору имена макросов, то к этим именам рекурсивно применяется макроподстановка до получения строки, не содержащей имена макросов. Например, после обработки препроцессором нижеследующего участка кода строка

`printf ("DEBUG\n");` будет вставлена 16 раз.

```
#define ONE printf ("DEBUG\n")
#define TWO ONE; ONE
#define FOUR TWO; TWO
#define SIXTEEN FOUR; FOUR; FOUR; FOUR
SIXTEEN;
```

Выше мы рассматривали макросы, тела которых всегда раскрываются одинаково.

Макроподстановки можно варьировать, задавая формальные *аргументы макроса* (в скобках после имени макроса) и используя их в теле макроса. При макроподстановке вместе с именем макроса указываются фактические аргументы в скобках, и в подставляемом теле макроса все вхождения формальных аргументов заменяются на фактические. При этом фактические аргументы макроса могут быть любыми последовательностями символов или выражениями Си.

Например, описанный макрос `DEBUGPRINT` можно усовершенствовать, введя аргумент

`X: #define DEBUGPRINT(X) fprintf (stderr, "debug value:%d\n", X)` и используя его как `DEBUGPRINT (i+5);`

(будет создана строка `fprintf (stderr, "debug value:%d\n", i+5);`).

Необходимо подчеркнуть, что препроцессор выполняет исключительно текстовые подстановки, не заботясь о содержимом фактических аргументов. Например, для макроса

```
#define HALF(X) X/2
```

макроподстановка для строки `y = HALF (z + 4);`

будет выглядеть как `y = z + 4/2;`, при этом из-за разных приоритетов деления и сложения предполагаемое значение `y` будет неверным. Во избежание таких ситуаций и для четкой расстановки приоритетов операций необходимо заключать в скобки как тело макроса, так и все вхождения в него формальных аргументов:

```
#define HALF(X) ((X)/2).
```

Кроме того, необходимо избегать написания макросов, в телах которых аргументы вычисляются более одного раза, а также передавать в макросы в качестве фактических аргументов выражения с побочным эффектом. Например, для макроса

```
#define ABS(X) ((X) >= 0 ? (X) : -(X)),
```

вычисляющего модуль своего аргумента, использование его как `ABS(y--);` приведет к тому, что побочный эффект выражения `y--` будет вычислен два раза. Следовательно, пользоваться макросами необходимо с осторожностью, избегая их бездумного применения, которое может быть заменено функциями. Основным сценарием использования является автоматическая генерация однотипного кода, который нежелательно писать вручную, или обеспечение переносимости программы (см. ниже).

Некоторые макросы уже определены компилятором Си, и их можно использовать в любой программе. Как правило, такие макросы содержат информацию либо о свойствах компилятора, например, о поддерживаемой им версии Си (макрос `__STDC_VERSION__`), либо о свойствах компилируемого файла и окружения и используются при отладке, например, макросы

`__FILE__, __LINE__, __DATE__` и другие.

Каждый компилятор часто добавляет собственные заранее определенные макросы к стандартным. Например, для компилятора GCC узнать полный список таких макросов можно, запустив команду препроцессирования пустого файла с записью макросов:
`gcc -dM -E - < /dev/null`.

Кроме этого, препроцессор также позволяет организовать включение определенных частей файла в окончательный текст файла в зависимости от некоторых условий. Чаще всего эта возможность служит для повышения переносимости программы. При первоначальной настройке программы на конкретную архитектуру компьютера с помощью ряда тестов создается конфигурационный заголовочный файл, в котором с помощью директив `#define` заводятся имена-макросы, означающие наличие или

отсутствие определенных свойств архитектуры или операционной системы. Директивы #ifdef/#else/#endif позволяют проверить, определено ли конкретное имя, и в зависимости от этого включить или не включить текст между #ifdef и #endif в окончательный текст файла. Вариант с директивой #else позволяет включить альтернативный текст при невыполнении условия. Например, если в конфигурационном файле есть определение #define HAVE_LONG_DOUBLE 1, то следующий фрагмент кода позволяет переносимо определить тип longest_float:

```
#ifdef HAVE_LONG_DOUBLE
  typedef long double longest_float;
#else
  typedef double longest_float;
#endif
```

Другим важным случаем использования условной компиляции является защита от двойного включения заголовочного .h файла в .c файл реализации. Заголовочные файлы содержат объявления переменных и функций, которые могут привести к ошибкам в случае их неоднократного включения в текст программы. Для большой программы со сложными связями между отдельными заголовочными файлами ситуация двойного включения одного файла, возможно, через другие заголовочные файлы, может легко возникнуть. Во избежание такой ошибки обычно используют следующий прием: для каждого заголовочного файла определяют уникальное имя макроса, и весь текст файла подключается лишь при условии, что данное имя не определено. Например, для стандартного файла stdio.h:

```
#ifndef _STDIO_H
#define _STDIO_H
<... текст файла ...>
#endif
```

11.2. Компоновщик

Компоновщик обеспечивает сборку программы в едином исполняемом файле из набора объектных файлов. Для задания правил компоновки глобальных переменных и функций используются понятия *внутренней* и *внешней* компоновки (третья ситуация, без компоновки, используется для локальных и регистровых переменных, которые не могут быть видимы за пределами текущего файла). Глобальные переменные, объявленные с квалификатором static, подлежат внутренней компоновке – их имена должны быть уникальны в пределах файла. Глобальные статические переменные с одинаковым именем в разных файлах разрешены, при компоновке получат разные области статической памяти и могут использоваться независимо.

Глобальные переменные, объявленные без квалификатора или с квалификатором extern, подлежат внешней компоновке. Во всех компонуемых файлах конкретное имя может иметь лишь одна глобальная внешняя переменная. При этом такая переменная может быть объявлена в нескольких файлах, обязательно с квалификатором extern. Эти объявления не выделяют памяти для переменной, но лишь уведомляют компилятор о

наличии в статической памяти переменной с определенным типом и именем. Определением глобальной внешней переменной является ее объявление без квалификатора `extern`, такое определение может содержать инициализацию переменной константным выражением и должно быть единственно для всех компонуемых файлов. Такое определение выделяет необходимый для переменной объем статической памяти, и все объявления этой глобальной переменной с квалификатором `extern` ссылаются на этот участок памяти. Таким образом, обеспечивается использование одной переменной разными файлами программы.

Функции, как и переменные, должны быть объявлены без квалификатора `static` или с квалификатором `extern` для того, чтобы их можно было вызывать из других файлов (т.н. *внешние* функции). По соглашению функции, определенные в других файлах, объявляются с квалификатором `extern` и, как правило, включаются в соответствующие заголовочные файлы. Среди компонуемых файлов может быть лишь единственное определение (но много объявлений) внешней функции с данным именем. Функция же, объявленная или определенная с квалификатором `static`, является статической и невидима для функций из других файлов. Тем не менее, например, она может быть вызвана через указатель, переданный в функцию из другого файла. Разрешено определять статические функции с одинаковым именем в разных программных файлах.

Хорошей практикой программирования является объявление функций и глобальных переменных, которые не нужны для других программных файлах, как статических (с использованием квалификатора `static`). Только функции и переменные, которые составляют интерфейс данного файла, должны быть объявлены как внешние.

11.3. Понятие об отладке. Отладчик gdb

Отладкой называется процесс поиска и удаления ошибок из программы. Как правило, при отладке известны входные данные, на которых программа работает некорректно или генерирует неверные выходные данные. Основная сложность заключается в обнаружении причины неверного поведения – набора точек программы, в которых данные обрабатываются неправильно, что ведет к неверному выходу программы. Собственно исправление найденных ошибочных мест программы часто требует меньше усилий.

Базовым способом отладки является т.н. *отладочная печать* – вставка в программу операторов, выводящих информацию о текущем значении переменных программы и принимаемых ею решениях. Отладочную печать принято выполнять на стандартный поток ошибок. Для удобного включения/отключения отладочной печати во всей программе выдачу печати обычно контролируют единым макросом, опцией командной строки или переменной окружения. Часто используют разные уровни детализации информации при отладочной печати, также управляемые опциями командной строки или переменными окружения.

Основным инструментом отладки программ являются *отладчики*. Отладчик позволяет получать информацию о поведении программы на заданном наборе входных данных, не меняя (в идеальном случае) ее поведения. Как правило, отладчик запускает программу на заданных входных данных в управляемом режиме, позволяя в любой момент времени остановить или возобновить выполнение, пошагово (с точностью до отдельных строк кода на Си или ассемблере целевой машины) выполнять программу, останавливать выполнение по достижении заданных точек программы (т.н. *точки останова*) безусловно или при выполнении некоторого условия на значения заданных

ячеек памяти, останавливать выполнение при изменении значения в заданных ячейках памяти (т.н. *точки наблюдения*), просматривать или изменять (во время остановки выполнения) значения переменных, ячеек памяти, просматривать текущий стек вызовов.

Нужно отметить, что отладчик управляет программой в бинарном виде (машинных кодах и ячейках памяти), а программисту требуется получать информацию в терминах программы языка Си (строк и операторов исходного кода программы, переменных). Для обеспечения такой возможности компиляторы Си позволяют генерировать т.н. *отладочную информацию*, которая устанавливает соответствие между объектами программы на языке Си и объектами машинной программы, и записыать ее в специальные части исполняемого файла программы. Без отладочной информации пользование отладчиком возможно только на уровне языка ассемблера. В компиляторе GCC выдача отладочной информации включается опцией командной строки `-g`. Рекомендуется также отключать выполнение оптимизаций компилятора с помощью опции `-O0`, так как построение и поддержание отладочной информации в ходе проведения оптимизаций компилятором является весьма нетривиальной задачей и плохо поддерживается компиляторами.

Стандартным отладчиком для Unix-подобных систем является отладчик GDB. Этот отладчик является инструментом командной строки, то есть управление отладчиком осуществляется введением команд с клавиатуры. В рекомендованной для проведения практикума среде Code::Blocks реализована поддержка отладчика GDB в графическом интерфейсе, то есть управление отладчиком возможно с помощью мыши и горячих клавиш, а информация отладчика выводится в отдельных графических окнах и в редакторе исходного кода программы. Подробнее об этом можно узнать в документации к среде на сайте http://wiki.codeblocks.org/index.php?title=Debugging_with_Code::Blocks.

Часто используемые команды GDB показаны в таблице.

Команда	**Описание**
`gdb <file> --args <args>`	загрузить программу с заданными параметрами командной строки
`run/continue`	запустить/продолжить выполнение
`break <function name/ file:line number>`	завести безусловную *точку останова*
`cond <bp#> condition`	задать условие остановки выполнения для некоторой точки останова
`watch <variable/address>`	задать *точку наблюдения* (остановка выполнения при изменении значения переменной или памяти по адресу) [2]
`next/step`	выполнить текущую строку исходного кода программы без захода/с заходом в вызываемые функции
`print <var>/ set <var> = expression`	посмотреть/изменить текущие значения переменных, памяти
`bt`	посмотреть текущий стек вызовов

[2] Точки наблюдения рекомендуется задавать по именам глобальных переменных или по адресу, т.к. точка наблюдения за локальной переменной становится некорректной сразу по выходе переменной из области видимости.

ЛИТЕРАТУРА

1. Б. Керниган, Д. Ритчи. Язык программирования С. Второе издание. Издательский дом “”Вильямс, 2010.

2. Stephen Prata. C Primer Plus. Fifth Edition. Sams Publishing 2004. ISBN 0-672-32696-5.

3. А.А. Белеванцев, С.С. Гайсарян, В.П. Иванников, Л.С. Корухова, В.А. Падарян. Задачи экзаменов по вводному курсу программирования (учебно-методическое пособие). М.: Изд. отдел ф-та ВМК МГУ имени М.В. Ломоносова, 2012.

4. К.А. Батузов, А.А. Белеванцев, Р.А. Жуйков, А.О. Кудрявцев, В.А. Падарян, М.А. Соловьев. Практические задачи по вводному курсу программирования (учебное пособие). М.: Изд. отдел ф-та ВМК МГУ имени М.В. Ломоносова, 2012.

5. Т.В. Руденко. Сборник задач и упражнений по языку Си (учебное пособие). М.: Изд. отдел ф-та ВМК МГУ имени М.В. Ломоносова, 1999.

6. Н.Д. Васюкова, И.В. Машечкин, В.В. Тюляева, Е.М. Шляховая. Краткий конспект семинарских занятий по языку Си (учебно-методическое пособие). М.: Изд. отдел ф-та ВМК МГУ имени М.В. Ломоносова, 1999.

СОДЕРЖАНИЕ