

Ю.Н. Артамонов, В.А. Докучаев, С.В. Шевелев

МОДУЛЬНОЕ ПРОЕКТИРОВАНИЕ ПРОГРАММНЫХ ПРИЛОЖЕНИЙ

Учебное пособие

**Москва
Ай Пи Ар Медиа
2023**

УДК 519.683+004

ББК 32.973.2

А86

Авторы:

Артамонов Ю.Н. — д-р техн. наук, проф. кафедры
экономики городского хозяйства и жилищного права
Московского городского университета управления
Правительства Москвы имени Ю.М. Лужкова;

Докучаев В.А. — д-р техн. наук, проф., зав. кафедрой
сетевых информационных технологий и сервисов
Московского технического университета связи и информатики;

Шевелев С.В. — канд. техн. наук, доц. кафедры
сетевых информационных технологий и сервисов
Московского технического университета связи и информатики

Рецензенты:

Шаврин С.С. — д-р техн. наук, проф. кафедры
многоканальных телекоммуникационных систем

Московского технического университета связи и информатики

Мышенков К.С. — д-р техн. наук, проф. кафедры ИУ5 «Системы обработки
информации и управления» Московского государственного
технического университета имени Н.Э. Баумана

Артамонов, Юрий Николаевич.

А86 Модульное проектирование программных приложений : учебное пособие /
Ю.Н. Артамонов, В.А. Докучаев, С.В. Шевелев. — Москва : Ай Пи Ар Медиа,
2023. — 172 с. — Текст : электронный.

ISBN 978-5-4497-2116-7

В учебном пособии приведены основные сведения о языке программирования С, рассмотрены вопросы разработки различных численных алгоритмов, игровых программ с демонстрацией использования облачных сервисов и алгоритмов с динамическими структурами данных. Помимо теоретического материала издание содержит также практические примеры с подробным разбором и задания для самостоятельной работы.

Подготовлено с учетом требований Федерального государственного образовательного стандарта высшего образования.

Учебное пособие предназначено для бакалавров, магистрантов и аспирантов укрупненных групп направлений подготовки «Информатика и вычислительная техника», «Электроника, радиотехника и системы связи», изучающих дисциплины «Архитектура центров обработки данных», «Технологии и средства облачных сервисов», «Программно-конфигурируемая архитектура приложений и инфраструктуры», «Методы и средства проектирования информационных систем и технологий», «Технологии программирования мобильных приложений», «Технологии программирования Web-приложений».

Учебное электронное издание

ISBN 978-5-4497-2116-7

© Артамонов Ю.Н., Докучаев В.А.,
Шевелев С.В., 2023

© ООО Компания «Ай Пи Ар Медиа», 2023

Учебное издание

Артамонов Юрий Николаевич
Докучаев Владимир Анатольевич
Шевелев Сергей Владимирович

Редактор *А.Д. Талмаева*
Компьютерный набор и верстка в системе Latex *Ю.Н. Артамонов*
Корректор *Е.В. Савенкова*
Обложка *Я.А. Кирсанов, С.С. Сизиумова*, фотобанк «Лори»

Подписано к использованию 24.03.2023. Объем данных 9 Мб.

ООО Компания «Ай Пи Ар Медиа»
8 800 555 22 35 (бесплатный звонок по России)
E-mail: sale@iprmedia.ru

СОДЕРЖАНИЕ

1	Введение	6
2	Язык программирования СИ	9
2.1	Общие сведения	9
2.2	Основные управляющие конструкции в языке Си	18
2.3	Указатели, массивы, динамические структуры . .	55
2.4	Вопросы для самоконтроля	76
3	Практикум по разработке численных алгоритмов	84
3.1	Суммирование рядов и вычисление элементарных функций	84
3.1.1	Пример вычисления суммы ряда	84
3.1.2	Задания для самостоятельной работы . . .	91
3.2	Приближенные методы нахождения корней уравнения	95
3.2.1	Пример использования метода последовательных приближений	95

3.2.2	Другие методы приближенного нахождения корней уравнений	109
3.2.3	Задания для самостоятельной работы . . .	112
4	Практикум по разработке игровой программы	115
4.1	Пример разработки игровой программы	115
4.2	Система управления версиями программного обеспечения git	148
4.3	Задания для самостоятельной работы	166
	Литература	171

В настоящее время интерес к различным системам программирования существенно возрастает, что продиктовано расширением спектра решаемых задач с использованием техник программирования: простые вычислительные задачи с предъявлением требований высокого быстродействия, точности; аналитика данных с возможностью гибких форматов представления, обработки; облачные сервисы с обеспечением надежного и безопасного хранения данных; игровые программы и другие.

Как правило, под каждым классом задач уже закрепились определенные технологии и системы. Однако, рассматривая их отдельно друг от друга, мы теряем общую картину возможностей - полезно понимать, что, используя некоторый базовый набор технологий, можно успешно решать любые задачи. Руководствуясь этой центральной идеей, авторы учебного пособия предприняли попытку продемонстрировать концепцию модульного проектирования программных приложений, выбрав в качестве средства реализации программ язык программирования

С. Конечно, авторы осознают, что выбор носит полемический характер. Однако в свое оправдание можно указать на поразительную адаптивность этого языка, которая на славу послужила, например, при создании ядра операционной системы Linux.

Отметим, что излагаемый учебный материал в большей степени ориентирован именно на использование операционных систем семейства GNU/Linux, для которых язык С является фактически «родным».

Учебное пособие включает в себя три основные части:

1. основные сведения о языке программирования С;
2. разработка различных численных алгоритмов;
3. разработка игровых программ с демонстрацией использования облачных сервисов, разработки алгоритмов с динамическими структурами данных.

Большая часть теоретического материала подкрепляется подробным разбором решаемых задач на практике. В этом отношении учебное пособие можно одновременно рассматривать и как компьютерный практикум по изучаемым темам, для чего в учебное пособие добавлены практические задания. Предполагается, что обучающийся уже имеет некоторые навыки программирования на каком-либо языке.

Задания первой, второй частей учебного пособия базируются на знаниях, полученных студентами по смежным дисциплинам: «Математический анализ», «Линейная алгебра», «Дискретная математика». Реализация соответствующих алгоритмов на языке программирования С позволяет глубже понять суть подходов к решению задач, дополняет полученные теоретические сведения практическими навыками по принципу: «Чтобы полностью понять решение задачи, следует научиться решать ее компьютер».

Третья часть учебного пособия связана с реализацией некоторой игровой программы с ее поэтапным развитием по нескольким направлениям. Это стимулирует у студента творческий под-

ход к реализации заданий, данный материал может быть использован в рамках дисциплин: «Распределенные операционные системы», «Технологии и средства облачных сервисов», «Архитектура центров обработки данных».

В учебном пособии также рассмотрен следующий перечень вопросов: различные режимы компиляции семейства компиляторов gcc, отладчик gdb, использование системы символьных вычислений maxima, создание статических и динамических библиотек, использование библиотеки ncurses, применение утилиты make для компиляции сложных проектов, работа с системой управления версиями программного обеспечения git, а также хостинг git-репозитория на примере github.com.

В целом учебное пособие нацелено на развитие компетенций по разработке алгоритмов и программ, пригодных для практического применения в области информационных систем и технологий, созданию оригинальных алгоритмов и программных средств, в том числе с использованием современных интеллектуальных технологий, для решения профессиональных задач.

2.1 Общие сведения

В общем случае выделяют следующие группы языков программирования:

- Ассемблерные языки (Nasm, Tasm, Masm, Fasm)
- Языки высокого уровня (C/C++, Python, Java, Php, Haskell, Ruby, R, Scala, Erlang, Common Lisp, Pascal, Fortran и т.д.)

В то же время неформально можно говорить о следующей классификации: 1. для скорости; 2. для гибкости. И хотя всегда хочется найти золотую середину, в большинстве случаев очень интересны крайние случаи. Язык программирования C, а вслед за ним и C++ традиционно относят к классу «для скорости». На них пишут системное программное обеспечение, ядра операционных систем, драйвера аппаратных устройств. В этот же класс с еще большим приоритетом следует отнести ассемблерные языки, однако последние слишком детальны и аппаратно зависимы.

Фактически в них коды машинных команд лишь заменяются удобными для запоминания человеком *мнемониками*.

В конечном итоге программный код любого языка программирования должен быть *транслирован* в набор машинных команд, понятных компьютеру. В зависимости от того, как построен процесс трансляции, выделяют *интерпретируемые* и *компилируемые* языки программирования.

Для получения эффекта скорости в финальной версии программы используется именно *компиляция*, для достижения гибкости - *интерпретация*. Конечно, существуют и промежуточные варианты, претендующие на «золотую середину» - Java с трансляцией программы в байт-код, который затем выполняется в специальном окружении Java-машины. При интерпретации выполнение команд программы (перевод их в совокупность машинных кодов) происходит по отдельности с сохранением некоторого достигнутого состояния в рабочем окружении интерпретатора. Этим и достигается особая гибкость - возможность изменять, модифицировать уже работающую программу в режиме run-time.

В компилируемых языках программирования, к которым в частности относится рассматриваемый язык C, этапы создания исполняемой программы включают в себя: *предпроцессорную обработку, компиляцию, линковку*.

В ходе предпроцессорной обработки выполняются директивы препроцессора. Данные директивы применяются перед компиляцией и чаще всего связаны с выполнением команд некоторого простого макроязыка. В языке C такие команды отвечают, как правило, за добавление в текст компилируемой программы других файлов (заголовочных файлов с описанием функций сторонних библиотек), замену специальных символов в тексте программы на другие символы, настройку условной компиляции в зависимости от операционной системы.

Этап компиляции - это основной этап превращения программы на языке высокого уровня в код машинного языка. Итогом работы компилятора является *объектный код*. Это уже почти

исполняемая программа, однако в ней отсутствуют функции из используемых стандартных библиотек. Эти пробелы как раз заполняются на этапе линковки (компоновки). В результате компоновки создается *исполняемый образ*, в котором уже нет отсутствующих частей. Это собственно и есть исполняемая программа.

Прежде чем мы рассмотрим процесс компиляции программы на языке С, уместно дать краткую историческую справку по этому языку. Язык С родился в стенах Bell Laboratories, создателем языка является Деннис Ритчи. Первые версии появились в 1972 году как попытки написания операционной системы Unix. Перед конкурентами на тот момент язык С имел ряд преимуществ за счет использования строгой типизации, лаконичного синтаксиса. В течение следующих нескольких лет он получил широкое распространение и признание. Появилось много различных реализаций языка С, что в 1983 году инициировало процесс его стандартизации, который в 1989 году привел к появлению стандарта ANSI/ISO 9899:1990 (C90). На этом процесс стандартизации не завершился, в разные годы вносились дополнения и изменения. Так, некоторые синтаксические конструкции считаются вполне корректными с точки зрения более позднего стандарта C99 и нежелательными или недопустимыми с точки зрения C90. Этот факт важно понимать, при необходимости указывая, какому стандарту соответствует компилируемая программа.

Над популяризацией языка С активно работал и его создатель. В 1978 году была написана книга Кернигана и Ритчи «Язык программирования С» [1], которая в дальнейшем много раз переиздавалась. Популярности языку придавало и широкое распространение операционных систем семейства Gnu/Linux, ядро которых было реализовано на языке С. Причем создатель Linux Линус Торвальдс при удобном случае критиковал С++ и хвалил С. Все это позволяет утверждать, что и сегодня язык С остается популярным, а иногда и просто незаменимым.

Перейдем теперь к обсуждению практических аспектов использования языка С.

В учебном пособии мы будем использовать компилятор языка C, входящий в состав набора компиляторов GCC (GNU Compiler Collection). Нужно отметить, что GCC - это именно набор компиляторов, родившийся под эгидой сторонников свободного программного обеспечения. Для компиляции программы на языке C следует сохранить текст программы с расширением «.c» и компилировать его с помощью команды `gcc`, для компиляции программы на языке C++ ее нужно сохранить с расширением «.cpp» и компилировать командой `g++`. И в первом, и во втором случаях будет использован набор GCC.

В базовую поставку GCC входят такие программы:

- `libc6-dev` - заголовочные файлы стандартной библиотеки Си;
- `libstdc++6-dev` - заголовочные файлы стандартной библиотеки C++;
- `gcc` - компилятор языка программирования Си;
- `g++` - компилятор языка программирования C++;
- `make` - утилита для организации сборки нескольких файлов;
- `dpkg-dev` - инструменты сборки пакетов deb.

Для установки GCC в операционных системах семейства GNU Linux (дистрибутив Debian или Ubuntu) можно использовать следующую команду:

```
>apt-get install build-essential
```

В качестве текстового редактора для набора исходных текстов программ можно использовать любой из понравившихся: `emacs`, `nano`, `gedit`, `vim`.

Для операционной системы Windows рекомендуется использовать Unix-подобную среду Cygwin, скачав ее с официального сайта: <https://cygwin.com/>

В качестве текстового редактора можно использовать программу расширенного блокнота Windows: Notepad++, которая доступна по ссылке: <https://notepad-plus-plus.org/>

Рассмотрим процесс компиляции на примере простейшей программы: требуется написать программу приближенного вычисления суммы ряда с точностью ε :

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

Листинг соответствующей программы представлен ниже.

```
#include <stdio.h>
#include <math.h>
int main()
{
    float x, epsilon;
    printf("x="); scanf("%f", &x);
    printf("epsilon="); scanf("%f", &epsilon);
    float Y=1,S;
    S = Y;
    int i=0;
    while (fabs(Y) >= epsilon)
    {
        i++;
        Y = Y*x/i;
        S = S+Y;
    }
    printf("S=%f\n", S);
    return 0;
}
```

На рис. 2.1 представлен скриншот консоли при компиляции и запуске данной программы, названной `example.c`.

Для компиляции используется команда:

```
> gcc -Wall -g example.c -o myprog
```

Здесь `-Wall`, `-g`, `-o` – ключи компиляции. Ключ `-Wall` подключает вывод в консоль всех предупреждающих сообщений при компиляции, ключ `-g` добавляет в созданный исполняемый файл отладочную информацию, которую можно впоследствии использовать отладчиком `gdb`, ключ `-o` применяется для указания имени созданного исполняемого файла (в данном случае использовано имя «`myprog`»). В самом упрощенном случае команда компиляции может выглядеть так:

```
> gcc example.c
```

Тогда созданный в результате ее выполнения исполняемый файл будет иметь имя по умолчанию (обычно это «`a.out`»).

Для запуска созданной программы на выполнение используется команда:

```
> ./myprog
```

или

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

для имени по умолчанию.

Кроме перечисленных ключей, `gcc` имеет очень большое количество других, с их полным составом можно познакомиться в справочной системе `man`, выполнив команду:

```
> man gcc
```

Перечислим здесь некоторые другие важные ключи [8]:

- `-c` - компилирует или ассемблирует исходный код, но не линкует его.
- `-S` - останавливает процесс компиляции на этапе получения ассемблерного кода, режим полезен для оптимизации кода.
- `-E` - останавливает процесс компиляции на этапе макропроцессирования, режим можно использовать для анализа корректности макроопределений.

```
Файл Правка Вид Поиск Терминал Справка
juna@artamonov:~$ gcc -Wall -g example.c -o myprog
juna@artamonov:~$ ./myprog
x=3
epsilon=0.001
S=20.085470
juna@artamonov:~$ ls -l myprog
-rwxr-xr-x 1 juna juna 11000 окт 26 10:14 myprog
juna@artamonov:~$ gcc -s example.c -o myprog
juna@artamonov:~$ ls -l myprog
-rwxr-xr-x 1 juna juna 6120 окт 26 10:16 myprog
juna@artamonov:~$ gcc -S example.c -o myprog
juna@artamonov:~$ cat myprog
        .file "example.c"
        .text
        .section      .rodata
.LC0:
        .string "x="
.LC1:
        .string "%f"
.LC2:
        .string "epsilon="
.LC5:
        .string "S=%f\n"
        .text
        .globl main
        .type  main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $32, %rsp
        movq    %fs:40, %rax
        movq    %rax, -8(%rbp)
        xorl    %eax, %eax
```

Рис. 2.1. Пример компиляции и использования программы

- `-l` - позволяет подключить к программе библиотеку функций, например, если требуется подключить библиотеку математических функций, необходимо задать ключ `-lmath` (или сокращенно `-lm`), если требуется использовать сокет, необходимо указать ключ `-lnsl`.
- `-L` добавляет директорию, в которой можно искать файлы библиотек, подключенных с помощью ключа `-l`.
- `-O n` - задает уровень оптимизации кода, по умолчанию $n=0$, что означает отсутствие оптимизации. Для более эффективного объектного кода рекомендуется использовать ключ `-O2`.
- `-s` - предписывает компилятору и линкеру исключить из кода всю вспомогательную, отладочную информацию. Данный ключ имеет смысл использовать на этапе финальной сборки.

В качестве примера на рис. 2.1 показан режим компиляции программы с разными ключами компиляции: `-g`, `-s`. Как видно из рисунка, при добавлении отладочной информации (ключ `-g`) объем исполняемого файла составил 11000 байт, а при исключении всей отладочной информации (ключ `-s`) объем уменьшился до 6120 байт. Также на рис. 2.1 продемонстрирован режим компиляции до ассемблерного кода (ключ `-S`), фрагмент которого выведен в консоль.

При разработке сложных программ часто требуется режим отладки, который позволяет выполнять программу по шагам с предоставлением возможности анализа значений переменных программы и используемых в ней функций. В этом случае удобным средством является отладчик (debugger) `gdb`.

`gdb` (GNU Debugger) - наиболее популярный отладчик `unix`-подобных операционных систем. Для его установки в системах семейства GNU Linux (дистрибутив Debian или Ubuntu) можно использовать следующую команду:


```
> apt-get install gdb
```

Отладчик gdb можно использовать как интерпретатор анализируемой программы, а также подключаться к уже запущенной программе. Чтобы использовать gdb в режиме интерпретации можно выполнить команду:

```
> gdb myprog
```

После данной команды мы попадаем в режим интерпретации программы myprog. Для пошагового запуска программы myprog используется команда **start**, чтобы выполнить следующий шаг исполняемой программы, нужно использовать команду **next**, чтобы прервать режим выполнения, нужно нажать **Ctrl+C**, а чтобы обратно продолжить выполнение, можно использовать команду **cont**.

Для того чтобы посмотреть значение какой-либо переменной, можно использовать команду **inspect имя переменной**, также можно принудительно присвоить ей значение с помощью команды **set var имя переменной=значение**.

Если требуется остановить программу на определенной строке кода, то номер строки можно получить командой **list** или **list номер строки** - выводит текст программы с заданной строки, добавляя несколько строк верхнего окружения.

Создать точку останова можно командой **break номер строки**. Чтобы запустить программу на выполнение до точки останова, можно использовать команду **run**.

Чтобы посмотреть перечень точек останова, следует использовать команду **info breakpoints**. Отключить точку останова можно командой **disable номер точки**, включить ее обратно – **enable номер точки**. Команда **ignore номер точки число** позволяет проигнорировать точку останова заданное число раз.

С помощью команды **call имя функции(значения параметров)** в gdb можно вызвать любую функцию, а с помощью команды **bt** посмотреть, какие функции, из каких мест программы и с какими параметрами были вызваны и выполняются сейчас.

Чтобы выйти из gdb, нужно использовать команду **quit**. Более подробную информацию о режимах отладки и используемых командах можно получить в справочных системах:

```
> man gdb  
> info gdb
```

На рис. 2.2 показан пример использования отладчика gdb для программы `myprog`.

2.2 Основные управляющие конструкции в языке Си

Как уже отмечалось, в учебном пособии не ставится цель подробного описания всех элементов программирования на языке Си для новичка. Для этого написано огромное количество замечательных книг, среди которых авторы особенно рекомендуют [1, 6]. И все же для более полного изложения в данном параграфе дается описание основных управляющих конструкций на языке Си. В целом синтаксис языка Си похож на многие другие языки, или, правильнее сказать, популярных Си-подобных языков довольно много: C++, Java, C#, Javascript и др. Сама программа представляет собой совокупность функций - блоков кода, которые получают на вход некоторые данные определенных типов, как-то их обрабатывают и возвращают результаты какого-то типа на выходе в то место, откуда функции были вызваны. Допустимо поведение, когда функции и ничего не возвращают. Среди всех функций есть особенная функция с именем `main` - это точка входа в программу. Иными словами, выполнение любой программы начинается с последовательного выполнения операторов функции `main`, из которой при необходимости могут быть вызваны другие функции. Такая функция в программе может быть только одна, если в коде отсутствует функция `main`, то он будет интерпретироваться как некоторая библиотека, из которой эти функции можно использовать.

```
Файл Правка Вид Поиск Терминал Справка
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from myprog...done.
(gdb) start
Temporary breakpoint 1 at 0x722: file example.c, line 4.
Starting program: /home/juna/myprog

Temporary breakpoint 1, main () at example.c:4
4      {
(gdb) next
6      printf("x="); scanf("%f", &x);
(gdb) next
x=6
7      printf("epsilon="); scanf("%f", &epsilon);
(gdb) inspect x
$1 = 6
(gdb) set var x=10
(gdb) inspect x
$2 = 10
(gdb) list
2      #include <math.h>
3      int main()
4      {
5          float x, epsilon;
6          printf("x="); scanf("%f", &x);
7          printf("epsilon="); scanf("%f", &epsilon);
8          float Y=1,S;
9          S = Y;
10         int i=0;
11         while (fabs(Y) >= epsilon)
(gdb) Quit
(gdb) cont
Continuing.
epsilon=
```

Рис. 2.2. Пример использования отладчика gdb

В языке Си запрещено внутри функции определять другие функции (использовать вложенные функции).

Рассмотрим в качестве примера простейшую программу типа «Hello world».

```
#include <stdio.h>

int main(){
    printf("Hello_world\n");
    return 0;
}
```

В программе используется одна функция `main`, внутри которой выполняются два действия - печать на экран сообщения с помощью функции `printf`, возврат из функции `main` с помощью оператора `return` с возвращением в качестве значения функции `main` целочисленного значения нуль. Для указания типа возвращаемого значения перед функцией `main` используется ключевое слово `int`, которое как раз означает целочисленный результат.

Как видно из листинга, операторы отделяются друг от друга точкой с запятой, блоки кода заключаются в фигурные скобки. Поскольку функция `main` не содержит входных данных, после имени функции идут пустые круглые скобки.

В самом начале программы используется специальная директива предпроцессора `#include`, которая отвечает за добавление в текст нашей программы на этапе предпроцессорной обработки прототипов функций из стандартной библиотеки ввода/вывода `stdio`. В частности функция `printf` - это функция стандартной библиотеки ввода/вывода. Аргументом функции `printf` является строка «Hello world», в конце строки за обратной косой чертой употребляется специальный ес-код `n`, отвечающий за перенос курсора на начало следующей строки экрана при последующем выводе на экран. Имеются и другие ес-коды:

`\t` - горизонтальная табуляция, перемещает курсор в следующую позицию табуляции;

`\r` - возврат каретки, перемещение курсора в начало текущей строки без перемещения на следующую строку;

`\\` - обратная косая черта, выводит на экран символ `\` при помощи `printf`;

`\"` - двойные кавычки, выводит на экран символ двойных кавычек при помощи `printf`.

Любая содержательная программа использует переменные, которые именуют некоторые области памяти, содержащие данные. Перед использованием переменной ее необходимо объявить, указав ее тип. Ниже представлен пример программного кода, демонстрирующий использование переменных в программе.

```
#include <stdio.h>

int main(){
    int a,b=1;
    float r,pi=3.14,S;
    printf("a=");
    scanf("%d",&a);
    printf("r=");
    scanf("%f",&r);
    b++;
    a=a+b;
    S=pi*r*r;
    printf("a=%d\tS=%f\n",a,S);
    return 0;
}
```

В приведенном примере объявляются две целочисленные переменные `a`, `b`, причем переменная `b` инициализируется начальным значением 1, также объявляются две вещественные переменные `r`, `pi`, переменная `pi` инициализируется начальным значением 3.14. Как видно из примера, чтобы объявить переменную,

нужно указать ее тип и при необходимости присвоить ей начальное значение. Также следует обратить внимание, что при выводе значений переменных на экран в функции `printf` указывается через запятую несколько аргументов. Первый аргумент имеет строковый тип, как, например, `a=%d\ts=%f\n`, причем, кроме ес-с-кодов `\t`, `\n`, там используются спецификаторы `%d`, `%f`. Они указывают `printf` тип переменных, которые должны быть выведены на месте этих спецификаторов. Спецификаторов может быть несколько, соответственно, после первого строкового аргумента должны быть перечислены все переменные, значения которых будут подставлены на место этих спецификаторов по порядку их следования.

В приведенном листинге также используется функция чтения информации с клавиатуры `scanf`. Формат данной функции похож на `printf`. Однако в первом строковом аргументе нужно использовать только спецификаторы, не отделяя их никакими символами. После перечисления спецификаторов следует указать адреса в памяти соответствующих переменных, которым будут присвоены введенные значения. Это выполняется использованием знака `&` перед именем переменной (подробно об этом будет рассказано в разделе про указатели).

В табл. 2.1 дано описание различных типов данных и соответствующих им спецификаторов для функций `printf`, `scanf`.

Из текста приведенной программы также можно увидеть сокращенное обозначение стандартной операции увеличения значения переменной на единицу: `c=c+1`, вместо такой длинной записи рекомендуется использовать выражение `c++` - операция *постинкремента*.

Рассмотрим более подробно эту возможность на примере следующей программы. Результат работы программы показан на рис. 2.3. Итак, в самом начале объявляются три переменные: `a`, `b`, `c`, причем переменная «`a`» инициализируется значением 1. Значения остальных переменных не определены, они указывают на какую-то область памяти, отведенную под эти переменные и содержащую любую информацию.

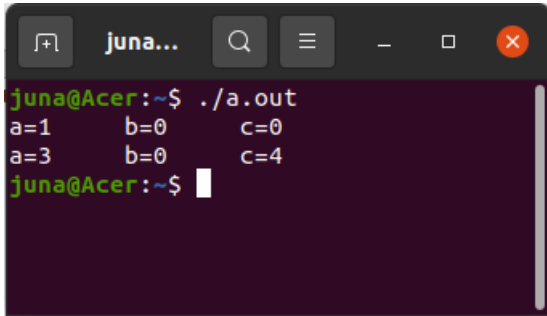
Таблица 2.1. Описание типов данных и их спецификаторов

Описание	Тип данных	printf	scanf
длинное вещественное двойной точности	long double	%Lf	%Lf
вещественное двойной точности	double	%f	%lf
вещественное	float	%f	%f
беззнаковое длинное целое	unsigned long int	%lu	%lu
длинное целое	long int	%ld	%ld
беззнаковое целое	unsigned int	%u	%u
целое	int	%d	%d
короткое целое	short	%hd	%hd
символьное	char	%c	%c

```
#include <stdio.h>
int main(){
    int a=1,b,c;
    printf("a=%d\tb=%d\tc=%d\n",a,b,c);

    a++;
    b=a;
    --b;
    c*=0;
    c+=(++a + b--);

    printf("a=%d\tb=%d\tc=%d\n",a,b,c);
    return 0;
}
```



```
juna@Acer:~$ ./a.out
a=1      b=0      c=0
a=3      b=0      c=4
juna@Acer:~$
```

Рис. 2.3. Результат работы программы, поясняющей операции инкремента, декремента

Из рис. 2.3, впрочем, видно, что в этой области памяти случайно содержатся нули, поскольку при выводе значений этих переменных напечатаны нулевые значения. Однако важно понимать, что неинициализированные переменные не гарантируют полученных значений каждый раз при запуске программы.

Операция `a++` увеличивает значение переменной на 1 (переменная «a» становится равной 2), причем это изменение происходит только после окончания текущего оператора, т.е. если бы в текущем операторе с переменной «a» проводились еще какие-то действия, то в них было бы использовано ее старое значение. Подобное поведение объясняет название этой операции - *постинкремент* (новое значение доступно только по окончании текущей операции).

Далее в строчке `b=a` новое значение переменной «a» присваивается переменной «b», после чего выполняется операция *преддекремента* (переменная «b» принимает значение 1). Операция `c*=0` – еще один из способов сокращенной записи операции `c=c*0`, данное действие обнуляет значение переменной «c».

Выражение `c+=(++a + b - -);` довольно сложная для понимания конструкция. Вначале с помощью операции *преинкре-*

мента значение переменной «a» увеличивается на 1 (было 2, станет 3), причем это новое значение тут же будет использовано в модификации значения переменной «с». Операция постинкремента переменной «b» приведет к ее уменьшению до нуля, но только после выполнения текущего оператора. В самом операторе модификации значения переменной «с» будет использовано старое значение переменной «b» (значение 1). В результате действие «(++a + b --)» дает значение 4, которое и будет присвоено переменной «с», инициализированной предварительно нулем.

В табл. 2.2 представлены основные формы сокращенной записи, используемые в С, а также дана их интерпретация. В табл. 2.2 предполагается, что переменные инициализированы следующим образом:

```
int a=1, b=2, c=3, d=5, e=4, f=6, g=12;
```

Таблица 2.2. Основные формы сокращенной записи на языке С

Операция присваивания	Пример выражения	Пояснение	Результат
++	a++ или ++a	a=a+1	a=2
--	b-- или --b	b=b-1	b=1
+=	c += 7	c = c+7	c=10
-=	d -= 4	d = d-4	d=1
*=	e *= 5	e = e*5	e=20
/=	f /= 3	f = f/3	f=2
%=	g %= 9	g = g%9 остаток от деления	g=3

Для реализации сложных алгоритмов, прежде чем писать код, часто удобно составить некоторую упрощенную схему (модель). В ней обычно абстрагируются от лишних подробностей, например не указывают тип переменных, специально не заботят-

ся об их объявлении. Для разработки такой упрощенной схемы наибольшее распространение получили такие способы, как псевдокод, блок-схемы.

Программы на псевдокоде помогают программисту «продумать» программу перед попыткой написать ее на каком-либо языке программирования. Вот пример алгоритма выбора наибольшего числа из двух чисел на псевдокоде:

```
1:  $n \leftarrow a, m \leftarrow b$   
2: if  $n > m$  then  
3:    $max \leftarrow n$   
4: else  
5:    $max \leftarrow m$   
6: end if
```

Псевдокод состоит только из операторов действия, в нем, как и было сказано, не используются объявления, предполагая, что программист сам знает, какая переменная что означает и какого она типа.

Блок-схемы - это так называемый графический способ записи алгоритмов. При разработке блок-схем используются некоторые символы специального назначения, такие как прямоугольники, ромбы, овалы и т.д. Все графические символы, их размеры, а также правила построения блок-схем определены государственными стандартами:

1. ГОСТ 19.701-90. ЕСПД. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения.
2. ГОСТ 19.002-80 ЕСПД. Схема алгоритмов и программ. Правила выполнения.
3. ГОСТ 19.003-80 ЕСПД. Схема алгоритмов и программ. Обозначения условные графические.

Для обозначения выполнения операции или последовательного выполнения группы операций, в результате чего изменяется значение, форма представления или расположения данных,

используется символ блок-схемы «*процесс*», который обозначается в виде обычного прямоугольника (рис. 2.4). Внутри символа или в виде комментария дается описание выполняемых действий на естественном языке.

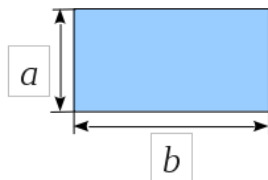


Рис. 2.4. Символ блок-схемы «Процесс»





Размеры символов должны удовлетворять соотношению $b = 1.5 \cdot a$, где a должен выбираться из ряда 10, 15, 20 мм. Допускается увеличивать размер a на число, кратное 5.

Для ввода-вывода данных в блок-схемах используются символы, представленные в табл. 2.3.

Для указания последовательности связей между символами используются *линии потока*. Для изображения линий потока существуют правила. Перечислим некоторые из них:

- линии потока должны быть параллельны линиям внешней рамки блок-схемы (границам листа, на котором изображена блок-схема);
- направление линии потока сверху вниз и слева направо принимается за основное и стрелками не обозначается, в остальных случаях направление линии потока обозначается стрелками;
- изменение направления линии потока производится под углом 90 градусов.

Таблица 2.3. Основные символы блок-схем для ввода-вывода

Название	Символ	Пояснение
Данные		Символ отображает данные, носитель данных не определен.
Ручной ввод		Ввод данных оператором в процессе обработки при помощи устройства, непосредственно сопряженного с компьютером (например, клавиатура).
Дисплей		Ввод-вывод данных в случае, если устройство воспроизводит данные и позволяет оператору вносить изменения в процессе их обработки.
Документ		Ввод-вывод данных, носителем которых служит бумага.

Изображение алгоритмов в виде блок-схем хорошо демонстрирует основные идеи структурного программирования, основоположниками которого принято считать итальянских исследователей К. Вёма и Д. Якопини. В своей статье Вём и Якопини доказали, что любую сколь угодно сложную программу можно представить чередованием лишь следующих трех управляющих структур:

- последовательной структуры (следование);
- структуры выбора;
- структуры повторения.

Последовательная структура, по существу, является встроенной в язык С. Если не указано иначе, компьютер автоматически выполняет операторы один за другим в порядке их записи. Нарисуем блок-схему следующей простейшей программы.

```
main()  
{  
    int a,b,c;  
    scanf("%d%d", &a, &b);  
    c = a+b;  
    printf("%d\n", c);  
    return 0;  
}
```

На рис. 2.5 показана соответствующая блок-схема.

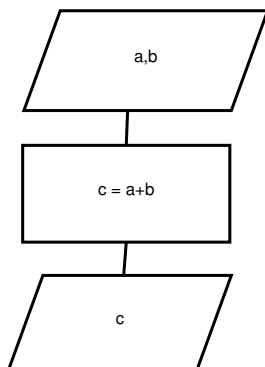


Рис. 2.5. Блок-схема программы, демонстрирующей структуры следования

Как видно из рис. 2.5, управление никуда не передается, команды выполняются строго последовательно. Данная структура описывает так называемые линейные алгоритмы: команды алгоритма выполняются последовательно друг за другом (линейно) без разветвлений и циклических повторений. Для всей совокупности последовательно выполняемых блоков часто строят обобщенную структуру в виде одного блока со входом и выходом, абстрагируясь от конкретного содержания этого блока (рис. 2.6).

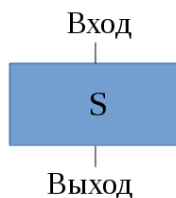


Рис. 2.6. Обобщенное представление автономного фрагмента блок-схемы со входом и выходом

Для графического изображения структуры выбора используется символ «решение» - выбор направления выполнения алгоритма в зависимости от некоторых условий (рис. 2.8).



Рис. 2.7. Символ блок-схем для обозначения структуры выбора

Выделяют следующие разновидности структур выбора:

- **Развилка полная.** Используется в случае, когда выполнение программы может пойти двумя различными путями. Внутри символа «Решение» (или в виде комментария к

этому символу) записывается логическое условие, по которому осуществляется выбор требуемого направления выполнения алгоритма. В зависимости от значения логического условия дальнейшее выполнение алгоритма идет либо по левой, либо по правой ветви. Символы «Процесс S1», «Процесс S2» могут обозначать унифицированные структуры, процедуры, функции и алгоритмы любой сложности (рис. 2.8).

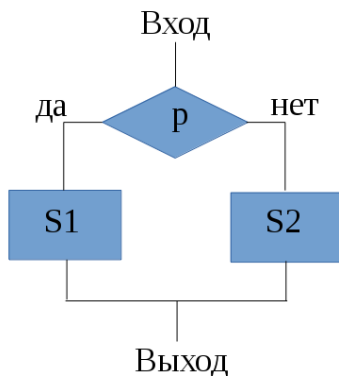


Рис. 2.8. Развилка полная в виде блок-схемы

- **Развилка неполная.** Используется так же, как и «развилка полная», с тем отличием, что при выполнении одной из ветвей никаких изменений данных, поступающих на вход этой унифицированной структуры, не происходит (рис. 2.9).
- **Выбор.** Предназначен для выбора из многих вариантов. Данную унифицированную структуру можно представить несколькими вложенными друг в друга структурами «развилка полная» (рис. 2.10).

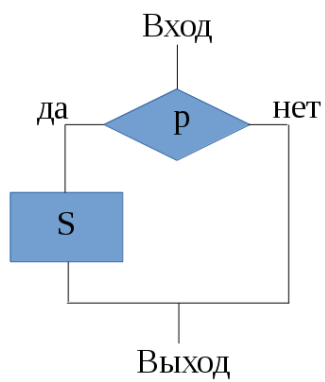


Рис. 2.9. Развилка неполная в виде блок-схемы

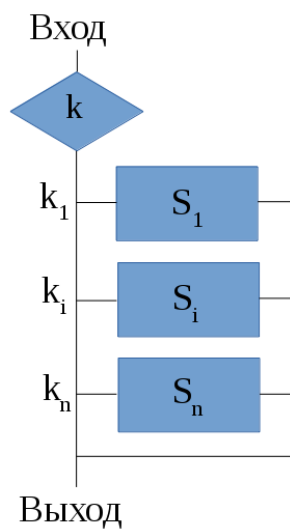


Рис. 2.10. Выбор в виде блок-схемы

Язык С предоставляет программисту все три типа структуры выбора:

- В структуре выбора **if** (развилка неполная) некоторые действия выполняются, если условие истинно, либо пропускаются, если это условие ложно.
- В структуре выбора **if/else** (развилка полная) некоторые действия выполняются, если условие истинно, и выполняется другое действие, если условие ложно.
- В структуре выбора **switch** (выбор) выполняется одно из набора различных действий в зависимости от значения некоторого выражения.

Рассмотрим использование этих условных конструкций на ряде примеров.

Предложим пример простой программы, которая загадывает случайное число из заданного диапазона и дает одну попытку его отгадать.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(){
    int N,P,R;
    srand(time(NULL));
    printf("Enter_the_number_N="); scanf("%d",&N);
    P=rand()%N+1;
    system("clear");
    printf("Guess_a_number_from_1_to_%d\n",N);
    printf("What_is_your_number?_");
    scanf("%d",&R);
    if (R==P) printf("You_guessed\n");
    printf("The_number_%d_was_guessed\n",P);
    return 0;
}
```

Для работы со случайными числами в С используется генератор псевдослучайных последовательностей. Для этих целей применяется библиотека `stdlib`. Номер псевдослучайной последовательности задается функцией `srand`, которая принимает некоторое целое число - номер последовательности. Однако при выборе одного и того же номера последовательности каждый раз будут получаться одинаковые псевдослучайные числа, что хорошо для повторяемости экспериментов, но не годится для различных игр. В этом случае номер последовательности выбирают, пересчитывая показания системных часов (функция `time(NULL)`) в номер последовательности. Благодаря такому действию в программе действительно номер последовательности выбирается случайно. Для получения очередного псевдослучайного числа из последовательности используется функция `rand()`. Она выдает число в диапазоне от 0 до значения специальной системной переменной `RAND_MAX-1`. Если требуется меньший диапазон значений, традиционно используют операцию получения остатка от деления на максимальное значение требуемого диапазона.

Также в коде используется вызов команды консоли Linux «clear» (очистить экран) с помощью оператора `system("clear")`. Сама условная конструкция выглядит предельно просто:

```
«if (R==P) printf("You guessed\n");»
```

после идентификатора «if» в круглых скобках пишется условие, в данном случае проверяются на равенство значения переменных «R» и «P». Наличие круглых скобок, обрамляющих условие, является обязательным, поскольку они выполняют роль разделения самого условия от операторов, выполняющихся, если условие верно. Если операторов после условия больше одного, их необходимо заключить в фигурные скобки.

На рис. 2.11 представлена блок-схема данной программы.

В качестве следующего примера, демонстрирующего универсальную структуру «Полная развилка», рассмотрим алгоритм нахождения максимального из трех чисел: пользователь вводит три числа, программа находит максимальное из них.

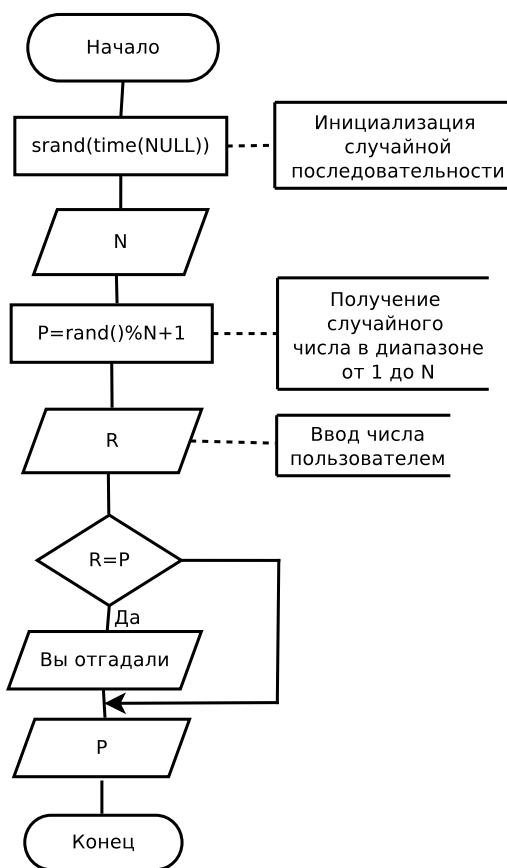


Рис. 2.11. Блок-схема алгоритма «Угадай число»

На рис. 2.12 показана блок-схема этого алгоритма. Как видно из блок-схемы, мы используем вложенные условные конструкции.

Ниже представлена соответствующая блок-схеме программа.

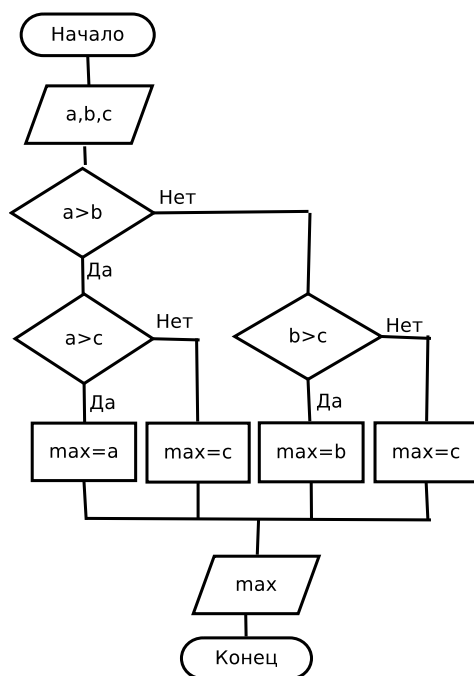


Рис. 2.12. Блок-схема алгоритма нахождения максимального из трех чисел

```

int main()
{
    int a, b, c, max;
    scanf("%d%d%d", &a, &b, &c);
    if (a > b)
    {
        if (a > c)
            max = a;
    }
}
  
```

```

        else
            max = c;
    }
    else if (b>c)
        max = b;
    else
        max = c;
    printf("%d\n", max);
}

```

Для реализации полной развилки в конструкции «if» после операторов, выполняемых при справедливости условия, используется ключевое слово «else», за которым пишут оператор, выполняемый при несправедливости логического условия. Если операторов больше одного, то они должны заключаться в фигурные скобки.

Для составления логических условий используют различные операции сравнения, а также логические связи, их назначение и примеры использования демонстрирует табл. 2.4.

В языке С за логическое значение «ложь» принято число ноль, любое другое числовое значение, отличное от нуля, воспринимается как логическая истина. В табл. 2.4 показано действие логических операций «или», «и» и «не», их приоритет меньше, чем у простых логических операторов, поэтому простые логические операции при их объединении в сложные логические выражения можно не заключать в скобки. Пример сложного логического выражения на языке С: $a > 100 \ \&\& \ b \neq 0$. Логическое отрицание имеет среди других логических операций наивысший приоритет. В большинстве случаев можно избежать использования логического отрицания в явном виде, например, условию $!(a!=0)$ соответствует $(a==0)$, а условию $!(b==0)$ условие $(b!=0)$.

В языке программирования С существует сокращенная запись инструкции if-else в виде условного выражения, результат которого можно даже присвоить переменной. Такая сокращенная форма имеет следующий вид:

(логическое выражение) ? выражение1 : выражение2

Если логическое выражение истинно, то такая логическая конструкция возвращает значение «выражение1» в качестве результата, если же логическое выражение ложно, то результатом станет значение «выражение2». Например:

```
a = 0;  
b = 1;  
z = (a > b) ? a++ : --b;  
w = (a!=b) ? a++ : b--;
```

В приведенном примере в первой условной конструкции условие $(a > b)$ оказывается ложным, поэтому выполняется предекремент переменной «b», и новое значение переменной $b=0$ присваивается в качестве значения переменной «z».

Во второй логической конструкции переменная «a» остается равна 0, а переменная «b» также стала равна нулю, т.е. условие $(a!=b)$ опять оказывается ложным, выполняется постдекремент переменной «b», но в качестве результата возвращается ее старое значение нуль, которое и присваивается переменной «w». В результате выполнения этого программного кода обе переменные «z,w» окажутся равны нулю. Обратите внимание, что после проверки условия вычисляется или выражение1, или выражение2, но не два одновременно.

В целом такая логическая конструкция бывает очень полезна, однако область ее применения ограничена простейшими случаями ветвления, т.к. при создании сложных условий использовать ее не всегда получается.

В качестве следующего примера продемонстрируем применение оператора выбора `switch`. Его использование оказывается удобным в случаях, когда требуется выполнить разбор целочисленных значений некоторой переменной и в зависимости от этих значений выполнить те или иные действия.

Структура `switch` состоит из ряда меток `case` и необязательного блока `default`, каждая метка заканчивается оператором `break`.

Таблица 2.4. Логические операции и операции сравнения
a=0,b=1,c=2

Название	Символ	Пояснения
Равенство	==	a==b False a==(b-1) True
Не равно	!=	a!=b True a!=a False
Больше	>	a>b False b>a True
Больше или равно	>=	(a+1)>=b True a>=b False
Меньше	<	a<b True (a+1)<b False
Меньше или равно	<=	a<=(b-1) True a<=b False
Логическое или		a a False a b True b c True
Логическое и	&&	a&&a False a&&b False b&&c True
Логическое не	!	!(a!=0) True !(a!=0)&&!(b==0) True

Для примера рассмотрим программу, которая анализирует код нажатой клавиши для организации экранного меню. Соответствующий код программы представлен ниже.

В данном случае программа реализует выполнение заданных арифметических операций сложения, разности, произведения, частного над введенными операндами в зависимости от выбранного действия.

```

#include <stdio.h>
#include <stdlib.h>
int main(){
    int s,a,b;
    printf("\t_Menu\n");
    printf("1._Addition\n");
    printf("2._Subtraction\n");
    printf("3._Multiplication\n");
    printf("4._Difference\n");
    s=getchar();
    system("clear");
    printf("a="); scanf("%d",&a);
    printf("b="); scanf("%d",&b);

    switch (s){
        case '1':
            printf("%d+%d=%d\n",a,b,a+b);
            break;
        case '2':
            printf("%d-%d=%d\n",a,b,a-b);
            break;
        case '3':
            printf("%d*%d=%d\n",a,b,a*b);
            break;
        case '4':
            printf("%d/%d=%f\n",a,b,(float)a/b);
            break;
        default:
            printf("Error\n");
            break;
    }
    return 0;
}

```


Для чтения кода нажатого символа используется функция `getchar()`, которая после нажатия клавиши ввода считывает первый символ со стандартного ввода. Следует обратить также внимание, что несмотря на то, что переменной «s» присваивается в результате такого действия символьный тип «char», сама переменная объявлена целого типа. Это связано с особенностью языка C, в котором символы кодировки ASCII представляются однобайтовым целым числом. Поэтому любое целое значение от 0 до 255 можно интерпретировать как символ, или наоборот, любой символ можно сохранить в целочисленной переменной. Например,

```
#include <stdio.h>
main()
{
    int s;
    s = getchar();
    printf("Symbol_%c, _ASCII-code:%d", s, s);
    return 0;
}
```

Здесь в первом случае значение целочисленной переменной «s» с помощью спецификатора %c интерпретируется как символ, а во втором случае эта же переменная спецификатором %d воспринимается как целое число.

Для реализации выбора пункта меню в приведенной программе с помощью оператора `switch` анализируется значение переменной «s». Для этого производится разбор четырех случаев. Например, если значение переменной «s» соответствует коду символа «1», то над введенными значениями переменных «a, b» выполняется операция суммы. Разбор случаев реализуется с помощью «case». После выполнения действий, прописанных в соответствующем «case», выполняется оператор `break`, который прерывает проверку остальных случаев и выполняет принудительный выход из оператора `switch`. Если `break` пропустить, то

после выполнения операторов соответствующего case проверка остальных условий продолжится, что не будет иметь никакого смысла, поэтому оператор break должен завершать действие каждого case. После описания всех возможных случаев часто добавляется блок default, операторы которого начинают выполняться, если ни один из case не сработал. Структура switch отличается от всех других структур тем, что в блоке case последовательность операторов не требуется заключать в фигурные скобки.

Ни одна содержательная программа не обходится без циклических конструкций. Выделяют следующие основные типы структур повторения:

- безусловный цикл (цикл со счетчиком);
- условный цикл с предусловием;
- условный цикл с постусловием.

В безусловном цикле некоторый блок операторов повторно выполняется заданное перед циклом количество раз. В блок-схемах для этих целей используется специальный символ «Подготовка», его внешний вид показан на рис. 2.13.

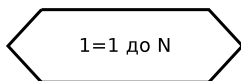


Рис. 2.13. Символ блок-схем «Подготовка» для реализации безусловных циклов

В качестве демонстрации использования безусловного цикла рассмотрим задачу нахождения суммы нечетных чисел от 1 до N. Блок-схема такого алгоритма представлена на рис. 2.14.

Как видно из рис. 2.14, для реализации безусловного цикла необходимо предусмотреть следующие составляющие:

1. переменную-счетчик; 2. начальное значение этой переменной; 3. значения инкремента (приращения со знаком плюс) или декремента (приращения со знаком минус), на которое переменная-счетчик изменяет свое значение после каждой итерации цикла; 4. условие, в котором происходит проверка на конечное значение управляющей переменной (т.е. должно ли продолжиться выполнение цикла).

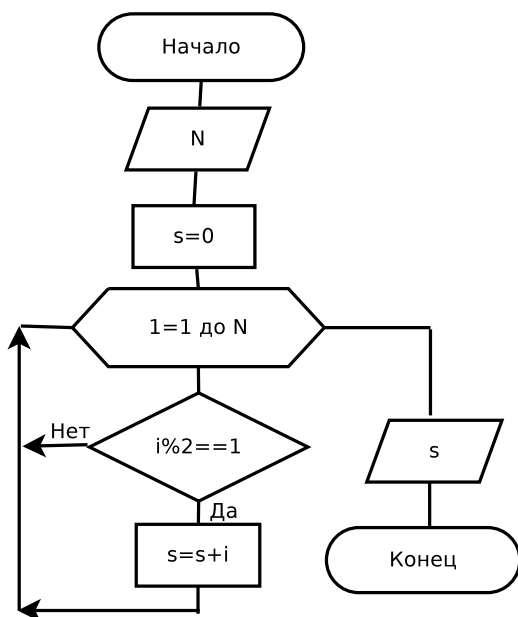


Рис. 2.14. Блок-схема алгоритма нахождения суммы нечетных чисел

Ниже представлена программа на языке С, соответствующая алгоритму рис. 2.14. В языке С для реализации безусловных циклов используется специальное ключевое слово «for». Структура оператора «for» включает в себя:

«for (Выражение1; Выражение2; Выражение3) оператор»

Выражение1 объявляет и инициализирует переменную цикла. Выражение2 является условием продолжения цикла. Выражение3 служит для приращения управляющей переменной.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int i,s=0,n;
    printf("n=");
    scanf("%d",&n);
    for (i=1;i<=n;i++)
        if (i%2==1) s+=i;
    printf("s=%d\n",s);
    return 0;
}
```

Заметим, что изменение управляющей переменной может быть не только в сторону увеличения, но и уменьшения, кроме того, шаг приращения может быть не только единичным. Например, если в нашей программе использовать шаг 2 с начального значения 1 управляющей переменной:

```
for (i=1;i<=n;i+=2) s+=i;
```

то можно обойтись без проверки условия и сделать программу более эффективной.

Однако язык С дает намного больше свободы при использовании оператора «for», допуская, что любое из выражений: Выражение1, Выражение2, Выражение3 может отсутствовать, также выражения могут состоять из нескольких операторов, разделенных запятой. Так, в рассмотренном примере можно оператор `s+=i` переместить из тела цикла в блок задания выражений:

```
for (i=1;i<=n;s+=i,i+=2);
```

Следующий фрагмент демонстрирует выполнение бесконечного цикла до тех пор, пока пользователь не введет комбинацию клавиш, означающую конец файла.

Мнемоника EOF (end of file) в операционной системе Linux соответствует комбинации клавиш «Ctrl+d», в операционной системе Windows - это комбинация клавиш «Ctrl+z».

```
for (;getchar()!=EOF);
```

Также в операторе for управляющую переменную цикла можно объявлять сразу внутри цикла:

```
for (int i=1;i<=n;i++)
```

Однако в этом случае переменная «i» имеет область видимости только внутри цикла (понятие «область видимости» рассматривается далее).

Условные циклы используются в ситуациях, когда заранее неизвестно, сколько итераций потребуется для выхода из цикла. Классическим в этом случае примером является расчет до достижения заданной точности. Например, требуется убедиться в справедливости тождества с заданной точностью:

$$\frac{3}{1 + \frac{3}{1 + \frac{3}{1 + \dots}}} - \frac{1}{3 + \frac{1}{3 + \frac{1}{3 + \dots}}} = 1$$

На рис. 2.15 показана блок-схема алгоритма, выполняющего проверку данного тождества, при этом используется цикл с предусловием.

Также ниже представлен код программы, соответствующей данной блок-схеме. Для реализации условного цикла с предусловием в языке C используется оператор «while», после которого в круглых скобках следует логическое выражение. Тело цикла while заключается в фигурные скобки после логического выражения и выполняется до тех пор, пока соответствующее логическое выражение возвращает значение «истина».

В представленной программе следует также обратить внимание на возможности форматированного вывода с помощью оператора printf. Для вывода значения first-second в программе используется спецификатор преобразования %20.10f.

Это означает, что будет выводиться число с плавающей точкой с точностью в 10 знаков после точки. Число 20 обозначает ширину поля, в которое будет выводиться значение. Если количество отображаемых символов меньше ширины поля, то значение автоматически выравнивается по правому краю. Чтобы выровнять значение по левому краю, нужно использовать знак минус. Например, для вывода значения переменной `n` как раз используется `%-10d`, что задает ширину поля в 10 позиций с выравниванием по левому краю.

В программе использована функция взятия модуля от вещественных чисел «`fabs`», входящая в состав математической библиотеки «`math`». Для ее подключения использован оператор «`#include math.h`». Кроме этого, для возможности использования функции математической библиотеки «`math`» программу следует компилировать с ключом «`-lm`».

```
#include <stdio.h>
#include <math.h>
int main(){
    double first , second , e;
    int n=0;
    printf("e=");
    scanf("%lf",&e);
    first=3; second=(float)1/3;
    while ( fabs(first-second-1)>e)
    {
        first=3/(1+first);
        second=1/(3+second);
        n++;
    }
    printf("Count\tValue\n");
    printf("%-10d\t%20.10f\n",n,first-second);
    return 0;
}
```

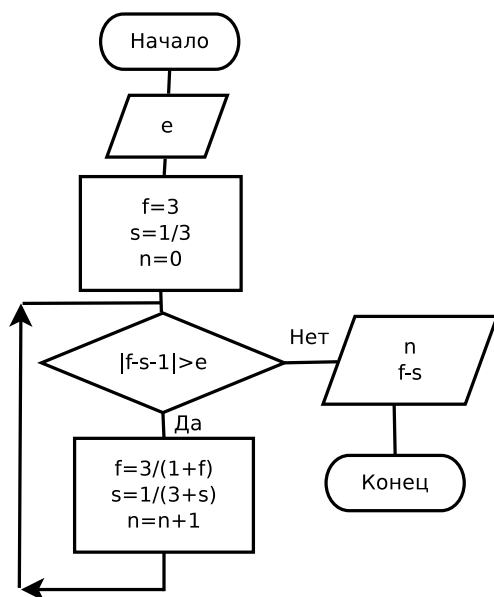


Рис. 2.15. Блок-схема алгоритма проверки тождества

Для реализации структуры повторения с постусловием в С используется конструкция `do/while`. В целом действие данной структуры подобно структуре повторения с постусловием. Отличие составляет то место, в котором проверяется логическое выражение. Если в цикле с предусловием проверка выполняется перед телом цикла, то в цикле с постусловием она выполняется после тела цикла. Поэтому в цикле с постусловием тело цикла выполняется как минимум один раз. Общий вид этого оператора:

`do { оператор } while (условие)`

Если оператор один, то фигурные скобки можно не ставить. Однако настоятельно рекомендуется их ставить всегда в этом

операторе, чтобы не спутать с оператором цикла `while`. Ниже представлен фрагмент той же программы с использованием цикла с постусловием.

```
do
{
    first=3/(1+first);
    second=1/(3+second);
    n++;
}
while ( fabs( first - second - 1 ) > e );
```

Кроме управляющих структур, для модульной разработки программных приложений необходимо предусматривать эффективные механизмы борьбы с повторением кода. Например, один и тот же фрагмент кода может требоваться в разных местах программы или даже в разных программах. Вместо его повторения целесообразно использовать модульный принцип - оформить повторяющийся код в виде отдельного модуля. Так мы приходим к понятиям функций и процедур, а также библиотечных модулей.

В блок-схемах для обозначения модулей-подпрограмм предусмотрен специальный символ «Предопределенный процесс», внешний вид которого показан на рис. 2.16.



Рис. 2.16. Символ «Предопределенный процесс»

Модули в языке `C` называются функциями. Программы обычно пишутся путем соединения новых функций, созданных программистом, с функциями, которые поставляются в составе *стандартной библиотеки C*. Хотя функции стандартной библиотеки технически не являются частью языка `C`, они неизменно поставляются с системами `ANSI C`.

Функции `printf`, `scanf`, `pow` являются функциями стандартной библиотеки. Программист может написать функции для решения своих задач, такие функции называются *функциями, определяемыми пользователем*. Обращение к функции осуществляется посредством *вызова функции*. При этом одна функция будет вызывать другую. Например, при использовании функции `printf` в своей программе на самом деле функция `main` осуществляет вызов функции `printf`. Обычно к функции обращаются, записывая её имя с последующей левой круглой скобкой, *аргументом функции* (или списком аргументов, отделяемых друг от друга запятыми) и правой круглой скобкой. Функции можно вкладывать одну в другую. Однако в языке C запрещено внутри определения одной функции определять другую функцию. Ниже в табл. 2.5 представлены функции стандартной математической библиотеки «math».

Именно функции позволяют программисту разбить программу на модули. Все переменные, объявленные в определениях функций, являются *локальными переменными* (они доступны только внутри функции, в которой определены). Большинство функций имеют список *параметров*. Параметры позволяют функциям обмениваться информацией. Параметры функции - это также локальные переменные. Продемонстрируем использование функций на примере проверки справедливости следующего соотношения:

$$\sum_{k=0}^{\infty} \frac{(-1)^k}{3k+1} = \frac{1}{9} (\sqrt{3}\pi + \log 8) \approx 0.83565$$

В целом следует придерживаться следующей концепции: необходимо стремиться реализовывать чистые функции, т.е. внутри функции не должно быть взаимодействия с пользователем средствами `printf`, `scanf`, функция не должна использовать глобальные переменные, которые можно изменить в основной программе, в каждый момент времени от одних и тех же аргументов функция должна возвращать одно и то же значение.

Таблица 2.5. Некоторые функции математической библиотеки
math

№	Обозначение	Назначение
1	sqrt(x)	квадратный корень из x
2	exp(x)	экспоненциальная функция e^x
3	log(x)	натуральный логарифм x (основание e)
4	log10(x)	десятичный логарифм x
5	fabs(x)	абсолютное значение x
6	ceil(x)	округление до ближайшего большего целого x
7	floor(x)	округление до ближайшего меньшего целого x
8	pow(x,y)	x возводится в степень y
9	fmod(x,y)	остаток от деления (fmod(6.5,3)=0.5)
10	sin(x)	тригонометрический синус от x в радианах
11	cos(x)	тригонометрический косинус от x в радианах
12	tan(x)	тригонометрический тангенс от x в радианах

Все интерфейсные взаимодействия с пользователем нужно выносить в основную функцию «main».

Вычисления правой и левой частей приведенного соотношения удобно оформить отдельными функциями. Ниже представлен соответствующий программный код. Как видно, в программе реализовано две функции: left и right, соответствующие левой и правой частям тождества. Функция left имеет на входе один формальный параметр «n» целого типа, определяющий количество итераций цикла. Сама функция возвращает вещественное значение (задается указанием типа перед именем функции). В качестве имени функции можно использовать любое слово, отве-

чающее правилам формирования идентификаторов, имен переменных. Внутри функции `left` определены локальные переменные «`s,c`», значения и имена которых остаются видны и доступны только в функции `left`. В другой функции, например в `main`, можно определить переменные с теми же именами, которые будут иметь совершенно другой смысл, тип, значение. Они также останутся доступны только внутри соответствующей функции.

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

float left(int);
float right(void);

float left(int n){
    float s=0,c=1;
    for (int i=0;i<=n;i++){
        s+=(float)c/(3*i+1);
        c*=-1;
    }
    return s;
}

float right(void){
    return (sqrt(3)*M_PI+log(8))/9;
}

int main(){
    int Q;
    printf("n=");
    scanf("%d",&Q);
    printf("%.16f\n",left(Q));
    printf("%.16f\n",right());
    return 0;
}
```

Также следует обратить внимание на так называемое приведение типов, выполняемое в строчке:

```
«s+=(float)c/(3*i+1);»
```

В языке С деление целых чисел дает целочисленный результат (целочисленное деление $3/2=1$). Для получения вещественного результата необходимо, чтобы хотя бы один аргумент имел вещественный тип. Этого можно достичь разными способами. В С традиционно используется приведение типов налету, для этого перед переменной в круглых скобках записывается приводимый тип: «(float) c». Правила приведения определяют, каким образом одни типы могут быть преобразованы в другие типы без потери данных. Однако такие преобразования возможны не всегда. Например, преобразование типа `double` в тип `int` отбрасывает дробную часть значения `double`. Правила возведения автоматически применяются к выражениям, содержащим значения двух и более типов данных (такие выражения называются *выражениями смешанного типа*). Каждое значение в выражении смешанного типа автоматически возводится к наивысшему типу выражения. Однако нужно иметь в виду, что процедура приведения типов применяется последовательно к каждой паре операндов. Например, вычисление $3/2*3.14$ даст в результате 3.14, поскольку 3 и 2 целые числа, и будет выполнено целочисленное деление. В табл. 2.1 приведены типы данных в порядке от наивысшего приоритета к низшему.

Кроме реализации функций `left` и `right`, в приведенном программном коде присутствует своеобразное описание структуры этих функций:

```
float left(int);  
float right(void);
```

Такое описание получило название *прототипа функции*. Данный подход заимствован стандартом ANSI C от языка C++. Основное назначение прототипов функций - это контроль типов аргументов функций, ее возвращаемых значений на этапе компиляции программы. Более ранние версии реализации С не поддерживали прототипы функций, и ошибки типов передавае-

мых в функцию аргументов выявлялись лишь на этапе выполнения программы в виде ее некорректной работы. Таким образом, компилятор использует прототипы функций для проверки корректности обращений к функции.

Использование прототипа не является обязательным, но его всегда целесообразно применять в программе.

Из приведенного примера также следует, что если функция ничего не возвращает, то перед ее именем нужно писать слово `void`. Если тип возвращаемого результата не указан, то компилятор по умолчанию будет считать, что это тип `int`. Тип результата в прототипе должен совпадать с типом результата при определении функции, иначе будет выдано сообщение об ошибке.

Существуют три способа возвращения управления в ту точку программы, из которой была вызвана функция. Если функция не возвращает результат, управление просто передается при достижении правой фигурной скобки, завершающей функцию, или при исполнении оператора `return;`. Если функция возвращает результат, оператор `return выражение;` возвращает вызывающей функции значение выражения.

Если функция должна что-то возвращать, а в ней `return` отсутствует, то это может привести к непредвиденным ошибкам! Хотя пропущенный тип по умолчанию расценивается как `int`, следует всегда задавать его явно.

Тип каждого параметра должен быть указан явно за исключением типа `int`. Если тип параметра не указан, то по умолчанию принимается тип `int`. Правило хорошего тона - включать тип каждого параметра в список параметров, даже если он относится к типу `int`, принятому по умолчанию. Повторное использование параметра функции как локальной переменной внутри функции приведет к синтаксической ошибке. Точка с запятой после правой круглой скобки, закрывающей список параметров в определении функции, является синтаксической ошибкой. При перечислении типов переменных в списке параметров необходимо указывать тип каждой переменной отдельно. Например, нельзя указать `float x,y`.

Это будет интерпретироваться так, что переменная «x» имеет тип float, а переменная «y» имеет тип по умолчанию int.

Кроме своего основного назначения, прототипы функции используются для создания библиотек. Для этого прототипы выносятся в отдельный файл, подключая его директивой `#include` по аналогии с подключением функций из стандартных библиотек.

Такой отдельный файл носит название *заголовочного файла*. Каждая стандартная библиотека имеет свой заголовочный файл, содержащий прототипы для всех функций данной библиотеки, а также определения различных типов данных и констант, необходимых этим функциям.

Перечислим имена некоторых заголовочных файлов стандартных библиотек:

`<math.h>` содержит прототипы функций математической библиотеки;

`<stdio.h>` содержит прототипы функций для ввода/вывода и информацию, используемую ими;

`<stdlib.h>` содержит прототипы функций преобразования чисел в текст и обратно, прототипы функций генерации случайных чисел и др.;

`<string.h>` содержит прототипы для функций обработки строк.

При создании своего заголовочного файла программист должен сохранить его с расширением `.h` там же, где находится сам файл программы. Данный заголовочный файл следует включать директивой: `#include "name.h"`. Здесь следует обратить внимание на отличия в подключении заголовочных файлов стандартных библиотек, где имя заголовочного файла заключается в угловые скобки. Использование угловых скобок указывает компилятору, что искать данные заголовочные файлы следует в специальных системных каталогах. Сама по себе директива `#include` предписывает дописать вместо директивы содержимое заголовочного файла. Таким образом, в файл просто дописываются прототипы используемых функций.

2.3 Указатели, массивы, динамические структуры

В предыдущем параграфе были рассмотрены основные управляющие структуры, по сути в том или ином виде присутствующие в любом языке программирования. Однако у языка программирования C есть своя специфика и продиктованная этим преимущественная область применения. В первую очередь язык C отличается от других языков уникальным низкоуровневым (физическим) доступом к памяти. Это делает язык C похожим на языки ассемблера с сопоставимой скоростью итоговой исполняемой программы и несопоставимо более удобным, гибким и лаконичным описанием необходимых действий в программе.

Ключевым понятием для этих целей служит понятие *указателя*. По сути указатель - это специальная переменная, значением которой является адрес памяти. Сам по себе указатель может хранить значение адреса памяти самых различных сущностей, которые используются в программе - переменных, массивов и даже функций. Чтобы получить адрес памяти какой-либо сущности, необходимо перед ее именем поставить знак амперсанда &.

Если говорят, что указатель указывает на переменную, то он содержит в качестве своего значения адрес памяти, где эта переменная хранит свое значение. Как и любую другую переменную, переменную-указатель нужно объявить перед использованием.

Различают два типа указателей - *типизированные* и *нетипизированные*. Типизированные указатели могут хранить адреса только определенных типов переменных. Тип указателя задается на этапе его объявления. Для объявления типизированной переменной-указателя вначале указывается тип, далее символ *, а затем имя указателя. При объявлении нетипизированного указателя вместо его типа пишется слово void.

Рассмотрим пример работы с указателями. В представленной ниже программе определены два типизированных указателя:

«р» (указатель на целочисленную переменную), «q» (указатель на вещественную переменную), а также один нетипизированный указатель «u». Указатели «р,q» неинициализированы при объявлении, указатель «u» инициализирован специальной константой NULL, обозначающей отсутствие значения. Указатель со значением NULL ни на что не указывает (символическая константа NULL определяется в заголовочном файле stdio.h). Присваивание значения 0 эквивалентно присваиванию NULL. Для того чтобы указатели связать с переменными, используются операции: «p=&n; q=&m». Важно понимать, что типизированная переменная-указатель должна указывать на переменную своего типа. Например, нельзя присваивать переменной-указателю «р» адрес вещественной переменной «m». Для нетипизированного указателя это ограничение не работает. Нетипизированному указателю можно присваивать адреса переменных совершенно разных типов.

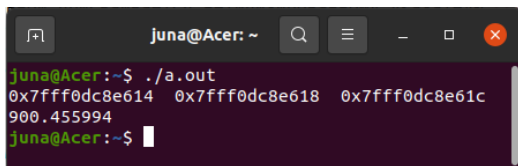
```
#include <stdio.h>

int main(){
    int n=777, *p;
    float m=123.456, s, *q;
    void *u=NULL;
    p=&n;
    q=&m;
    s>(*p + *q);
    u=&s;
    printf("%p\t%p\t%p\n",p,q,u);
    printf("%f\n",*((float*)u));
    return 0;
}
```

Для того чтобы получить значение переменной, на которую указывает переменная-указатель, выполняется *операция разыменования указателя*.

Для этого перед именем переменной-указателем ставится знак звездочки *. Таким образом, операции & и * взаимно дополняют друг друга. Так, в предложенной программе действие «s=(*p+*q)» приведет к разыменованию переменных «p, q», в результате чего переменной «s» будет присвоено вещественное значение «777+123.456».

Далее, как следует из кода, адрес переменной присваивается нетипизированному указателю «u». Для вывода значений переменных-указателей (адресов памяти, которые они содержат) в операторе printf используется специальный спецификатор «%p». Указатель типа void нельзя разыменовывать без приведения типов, поскольку заранее неизвестно, сколько байт памяти занимает тот объект, на который ссылается этот указатель. В то же время оператор «(float*)u» выполняет приведение типов, и сущность «(float*)u» становится указателем на вещественное число, который хранит адрес переменной «u». После этого операций «*((float*)u)» возможно разыменование полученного указателя. На рис. 2.17 показан пример работы приведенной выше программы. В первой строке выводятся три адреса памяти, которые являются значениями переменных-указателей «p, q, u» соответственно. В следующей строке выводится значение, на которое ссылается нетипизированный указатель «u».



```
juna@Acer: ~  
juna@Acer:~$ ./a.out  
0x7fff0dc8e614 0x7fff0dc8e618 0x7fff0dc8e61c  
900.455994  
juna@Acer:~$
```

Рис. 2.17. Пример работы с указателями

Особенностью работы с указателями является то, что они не только могут указывать на сущности, которые создаются в программе, и под них выделяется память в момент компиляции программы.

Память можно выделить на этапе выполнения программы, привязав ее к указателю.

Такой подход называется *динамическим выделением памяти*. Для динамического выделения памяти в языке C существует 2 различные функции:

```
void* malloc(Размер_в_байтах);  
void* calloc(Число_элементов, Размер_элемента_в_байтах);
```

Функции динамического выделения памяти находят в оперативной памяти непрерывный участок требуемой длины и возвращают начальный адрес этого участка. Обе функции в качестве возвращаемого значения имеют указатель на пустой тип void, поэтому требуется явное приведение типа возвращаемого значения.

Для определения размера одного элемента заданного типа рекомендуется использование функции sizeof(тип), которая определяет количество байт, занимаемое элементом указанного типа.

Память, динамически выделенная с использованием функций calloc(), malloc(), должна быть освобождена с помощью функции free(указатель). Если память не освободить, а потом потерять адрес начала этой области памяти (например, присвоив указателю другое значение), то этот объем памяти «повиснет в воздухе». Если же фрагмент этого кода, где выделяется и не освобождается память, выполняется в программе много раз, то возникает так называемый эффект «утечки памяти»: в ходе работы программы памяти, выделенной ей операционной системой, становится все меньше, даже если программа ничего не делает. Поэтому правилом хорошего тона в программировании является освобождение динамически выделенной памяти в случае отсутствия ее дальнейшего использования. В то же время, если динамически выделенная память не освобождается явным образом, она будет освобождена по завершении выполнения программы.

Рассмотрим пример на динамическое выделение памяти.

В представленной ниже программе выделяется 40 последовательно идущих байт (одно число типа `int` занимает 4 байта - `sizeof(int) = 4`) для хранения целых чисел, начало этой области памяти присваивается указателю «`p`». На рис. 2.18 показано, сколько байт памяти занимают разные типы данных.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(){
    int *p,*pp;
    float *f,*ff;
    p=(int*) malloc(10*sizeof(int));
    f=(float*) calloc(10,sizeof(float));
    pp=p;
    ff=f;
    srand(time(NULL));
    for (int k=1;k<=10;k++,pp++,ff++){
        *pp=rand()%100;
        *ff=(rand()%100)/(float)100;
    }
    ff=f;
    for (int i=0;i<=9;i++,ff++){
        printf("%d\t%f\n",*(p+i),*ff);
    }
    free(p); free(f);
    return 0;
}
```

Также в программе выделяется 40 байт (`sizeof(float)=4`) для хранения вещественных чисел, начало этой области памяти присваивается указателю «`f`». Далее в программе значения указателей «`p`, `f`» присваиваются указателям «`pp`, `ff`» соответственно для того, чтобы иметь возможность двигаться по памяти и не потерять начало выделенных областей.

```
short 2 bytes
int 4 bytes
unsigned int 4 bytes
long int 8 bytes
unsigned long int 8 bytes
long long int 8 bytes
float 4 bytes
double 8 bytes
long double 16 bytes
```

Рис. 2.18. Объем памяти, занимаемой разными типами данных

В цикле «for(int k=1;k<=10;k++,pp++,ff++)» области памяти, на которые указывают «pp, ff», заполняются случайными целыми и вещественными числами соответственно. Причем для перемещения к следующему адресу используется операция инкремента «pp++,ff++». Таким образом, можно сделать вывод, что над указателями можно выполнять некоторые арифметические операции, однако набор таких операций ограничен: указатель может быть увеличен (++) или уменьшен (--), к указателю может быть прибавлено целое число (+ или +=), из указателя можно вычесть целое число (- или -=), а также можно вычислить разность двух указателей. Стоит обратить внимание, что складывать указатели нельзя, поскольку полученный результат не будет иметь смысла.

Увеличение указателя на единицу или его уменьшение на единицу компилятор понимает не буквально, как с числами. При прибавлении или вычитании из указателя целого числа значение его увеличивается или уменьшается не на это число, а на произведение этого числа на размер объекта, на который указатель ссылается.

В нашем примере операции «pp++,ff++» будут увеличивать значения соответствующих указателей на 4, позволяя получить следующий адрес памяти, который можно заполнить случайным числом. Во втором цикле «for (int i=0;i<=9;i++,ff++)» на при-

мере указателя «р» продемонстрирован способ смещения по памяти с помощью индекса «р+і», это означает, что можно получать значение заданной ячейки памяти в цепочке последовательно идущих байт памяти по индексу, как в массивах. Фактически именно это свойство указателей и используется в языке С для реализации структуры - массив.

Однако, прежде чем перейти к массивам, рассмотрим еще ряд особенностей работы с указателями.

Вначале рассмотрим два важных понятия: *область видимости идентификатора* - это та часть программы, в которой возможно обращение к этому идентификатору, а также *время хранения идентификатора* - это время, в течение которого данный идентификатор существует в памяти.

Рассмотрим здесь такие области видимости: область видимости файла, область видимости блока. Идентификатор, объявленный вне любой функции, имеет область видимости файла. Такой идентификатор известен всем функциям, начиная с того места, где он объявлен, и до конца файла. Глобальные переменные, определения функций, прототипы функций, помещенные вне функций, - все они имеют область видимости файла.

Идентификаторы, объявленные внутри фигурных скобок, имеют область видимости блока. Такая область видимости заканчивается завершающей правой фигурной скобкой блока. Локальные переменные, объявленные в начале функции, имеют область видимости блока. Любой блок может содержать свои объявления переменных. Если блоки вложены, а идентификатор во внешнем блоке имеет такое же имя, как во внутреннем блоке, то виден только идентификатор внутреннего блока. Ниже приведен пример использования разных областей видимости.

Как видно из примера, после прототипов функций определена глобальная переменная «х» с областью видимости всего файла. Внутри функции main определяется своя переменная с именем «х» с областью действия блока. Данная переменная экранит глобальную переменную «х» внутри всего блока - тела функции main.

Также в `main` определяется локальная переменная «с», область ее видимости тоже видимость блока - тела функции `main`. Этой переменной присваивается значение «с=1». Однако далее в `main` используется вложенный блок, в котором опять определяется переменная «с», она экранирует область действия внешней по отношению к ней локальной переменной «с» внутри того блока, где она определена. Вызов функции «а» с переданным ей значением переменной «с» демонстрирует еще одну особенность: параметры по умолчанию передаются в функцию по значению, т.е. внутри функции мы работаем не с самой переменной «с», а с ее копией. Поэтому любые действия с этой переменной не приводят к изменению внешней по отношению к ней переменной «с» в функции `main`. В самой функции «а» выполняется инкремент формального параметра «х», который в данном случае принимает значение переданной переменной «с». Нужно обратить внимание, что переменная «х» внутри функции «а» также экранирует глобальную переменную «х». Во вложенном блоке функции `main` сначала печатается значение функции «а(с)», выполняющей инкремент «с» и соответствующей в данном случае числу 4, однако повторный вывод значения переменной «с» дает опять значение 3. Следующей строчкой после выхода из вложенного блока опять печатается значение переменной «с», но теперь вернулась область видимости тела функции `main`, где переменная «с=1», таким образом, печатается значение 1.

Вызов функции «b» демонстрирует, как в языке C добиться передачи параметра в функцию по ссылке. Для этого второй параметр функции «b» является указателем на целое число «int *х». При вызове функции «b» на место второго аргумента функции подставляется адрес памяти «&с», откуда в функции нужно брать значение. Поэтому изменение значения в памяти по адресу переменной «х» поменяет значение и в переменной «с», доступной из тела функции `main`. В данном случае переменной «х» присваивается ее же значение плюс значения переменной «с», переданной как копия. В итоге переменная «с» в блоке `main` получает значение «*х=*х+с=1+1=2».

Использование функции «сс» демонстрирует еще одну особенность работы с памятью: внутри нее определена локальная переменная с моделью памяти «static». Для этого перед типом переменной «x» используется ключевое слово «static».

```
#include <stdio.h>
int a(int);
void b(int, int *);
void cc(void);
int x;
int a(int x){
    x++;
    return x;}
void b(int c, int *x){
    *x=(*x)+c;}
void cc(void){
    static int x=1;
    printf("%d\n",x);
    b(x,&x);}
int main(){
    int c, x;
    c=1; x=2;
    {
        int c=3;
        printf("%d\n",a(c));
        printf("%d\n",c);}
    printf("%d\n",c);
    b(c,&c);
    printf("%d\n",c);
    {
        cc();
        cc();}
    return 0;
}
```

Локальные переменные, объявленные с ключевым словом `static`, остаются известными только из той функции, в которой они определены, но в отличие от обычных локальных переменных они сохраняют свое значение и после выхода из функции. При следующем вызове статическая переменная будет содержать то значение, которое она имела при последнем выходе из функции. Статические переменные целесообразно инициализировать перед использованием, если этого не происходит, они по умолчанию инициализируются нулем.

Как следует из действий внутри функции «сс», первый ее вызов из функции `main` приведет к печати значения 1. В то же время после печати за счет вызова функции «b» из «сс» значение переменной «x» инкрементируется и станет равно 2.

Повторный вызов функции «сс» пропускает инициализацию переменной «x», сохраняя ее старое значение, поэтому печать ее значения приведет к появлению числа 2. Таким образом, в результате выполнения программы на экране последовательно появятся следующие значения: 4; 3; 1; 2; 1; 2.

На данном примере видно, что если для обычных локальных переменных (их класс памяти обычно не указывается и обозначается ключевым словом `auto`) их время хранения ограничено временем выполнения соответствующего блока кода, где эта переменная объявляется и используется, то для глобальных и статических переменных это не так. Глобальные переменные по умолчанию объявляются с классом памяти `extern`, их время хранения начинается с момента их объявления и продолжается до момента окончания программы. Это же верно для переменных с классом памяти `static`: как только выполнение программы дошло до места объявления такой переменной, начинается отсчет времени ее хранения, которое аналогично завершается только по окончании работы программы. Говоря о классах памяти, следует еще упомянуть использование переменных цикла:

```
for (int i=0;i<10000;i++)...
```

При таком объявлении переменной «i» она остается видна только внутри блока цикла `for`. Если такую переменную хранить

в памяти, то при большом количестве итераций цикла придется каждый раз выполнять чтение ее значения из памяти. Чтобы сократить затраты времени на такие действия, компилятору можно рекомендовать загрузить ее в регистр процессора. Для этого переменная объявляется с классом памяти *register*:

```
for (register int i=0;i<10000;i++)...
```

Однако такая рекомендация остается на усмотрение компилятора: он может это сделать на этапе оптимизации кода сам даже при отсутствии ключевого слова *register* или, наоборот, проигнорировать такую рекомендацию.

Важным механизмом управления указателями является использование модификатора *const*. При обычном объявлении указателя ему можно присваивать разные адреса памяти (изменяемый указатель) и с помощью операции разыменования указателя менять значения по этим адресам памяти (изменяемые данные). Модификатор *const* позволяет добиться запрета либо изменения хранящегося в указателе адреса, либо запрета изменения значения переменной, на которую ссылается указатель, либо и того, и другого вместе. Таким образом, выделяют четыре отдельных случая:

изменяемый указатель на изменяемые данные:

```
int * p1;
```

неизменяемый указатель на изменяемые данные:

```
int * const p2;
```

изменяемый указатель на неизменяемые данные:

```
const int * p3;
```

неизменяемый указатель на неизменяемые данные:

```
const int * const p4;
```

В приведенной ниже программе демонстрируются допустимые методы работы с указателями при использовании разных ограничений доступа с помощью модификатора *const*.

```
#include<stdio.h>
int main(){
    int a=1,b=2,c=3,d=4;
    int * p1=&a;
    const int *p2=&b;
    int * const p3=&c;
    const int * const p4=&d;
    printf("a=%d\tb=%d\n",*p1,*p2);
    printf("c=%d\td=%d\n",*p3,*p4);
    (*p1)++;
    p2++;
    *p3=*p2;
    printf("a=%d\tb=%d\n",*p1,*p2);
    printf("c=%d\td=%d\n",*p3,*p4);
    return 0;
}
```

Указатель «p1» позволяет как поменять значение переменной «a», на которую он ссылается, так и сам может указывать на другой адрес памяти. Указатель «p2» можно поменять так, что он будет указывать на другой адрес памяти, например, как показано в программе «p2++» (поскольку память под переменные «a, b, c, d» выделялась последовательно, то операция «p2++», скорее всего, приведет к тому, что указатель «p2» станет ссылаться на переменную «c»). Однако менять значение переменной, на которую ссылается указатель «p2» (например, «(*p2)-»), нельзя. Указатель «p3», наоборот, позволяет изменить значение переменной, на которую он ссылается, но вот сделать так, чтобы он ссылался на другую переменную, не получится (например, операция «p3=p1» недопустима).

Наконец, для указателя «p4» запрещены любые модификации - нельзя поменять ни значение в памяти, ни адрес памяти. Такие ограничения доступа широко используются при передаче параметров в функцию по ссылке, а также при организации массивов.

В принципе работа с массивами - это такая же работа с указателями, просто для этого вводятся более удобные обозначения. Проведем сравнение на примере.

```
#include<stdio.h>
#include<stdlib.h>
int main(){
    int a[10],c;
    int * const p=(int *)malloc(10*sizeof(int));
    for (int i=0;i<10;i++){
        a[i]=i;
        *(p+i)=9-i;
    }
    for (int i=0;i<10;i++){
        c=*(a+i);
        *(a+i)=p[i];
        p[i]=c;
    }
    free(p);
    return 0;
}
```

В программе объявлен массив из 10 целых чисел. Память под такой массив выделяется уже при компиляции программы: в коде самой программы резервируются соответствующие последовательно идущие ячейки памяти, начальный адрес такой памяти присваивается переменной «a». Таким образом, переменная «a» по сути является таким же указателем. Правда, в кратком объявлении «int a[10]» завуалировано объявление «int *const a=(int*)malloc(10*size(int));», т.е. переменная массива «a»

является неизменяемым указателем с изменяемыми данными. Для сравнения ниже дано объявление аналогичного указателя «р», поскольку он тоже неизменяемый указатель, его объявление обязательно нужно сопрягать с его инициализацией, иначе далее это будет запрещено (ведь он также неизменяемый указатель).

Для разыменования указателя «а» используется более удобное обозначение - индексация «а[i]», причем первый элемент массива соответствует индексу нуль. В то же время, как видно из приведенной программы, для массива «а» остается доступным и ранее описанный способ перемещения по памяти с использованием приращения адреса: «а+і», и наоборот, для простого указателя «р» можно применять индексацию, как для элементов массива: «р[i]». Всё это раскрывает единство подхода в языке С: все динамические структуры реализуются в конечном итоге с использованием указателей.

Более сложно в языке С обстоит дело с реализацией многомерных (в частности, двумерных) массивов. Задать двумерный массив размерностью $n \cdot m$ с помощью указателей можно, создав массив из n указателей, каждый элемент которого ссылается на область памяти из m выделенных ячеек. Для этого необходимо:

- выделить блок оперативной памяти под массив указателей;
- выделить блоки оперативной памяти под одномерные массивы, представляющие собой строки искомой матрицы;
- записать адреса строк в массив указателей.

Рассмотрим пример, в котором матрица размерностью $n \cdot m$ заполняется случайными числами. Для выделения памяти под массив указателей, которые хранят адреса памяти строк матрицы, используется команда: `int ** a= (int **)malloc(n*sizeof(int *));`. Таким образом, переменная «а» становится указателем на область памяти, в которой хранят свои значения n указателей, которые, в свою очередь, будут хранить адрес памяти,

выделенной для m целых чисел, составляющих столбцы соответствующей строки. Для выделения памяти под m столбцов каждой из n строк используется команда: «*(a+i)=(int *)malloc(m*sizeof(int));».

Для навигации по такому двумерному массиву используется сначала индексация по адресу строки: «*(a+i)», а затем внутри строки индексация по столбцу: «*(*(a+i)+j)».

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#define n 3
#define m 4
int main(){
    int ** a= (int **) malloc(n*sizeof(int *));
    for (int i=0;i<n;i++)
        *(a+i)=(int *) malloc(m*sizeof(int));
    srand( time(NULL));
    for (int i=0;i<n;i++)
        for (int j=0;j<m;j++)
            *(*(a+i)+j)=rand()%100;
    for (int i=0;i<n;i++){
        for (int j=0;j<m;j++)
            printf("%d\t",*(*(a+i)+j));
        printf("\n");
    }
    for (int i=0;i<n;i++) free(*(a+i));
    free(a);
    return 0;
}
```

Важной особенностью работы с многомерными массивами является порядок освобождения памяти. В этом случае недостаточно освободить память только оператором «free(a)» - по такой команде мы освободим память от массива указателей на строки,

но память, выделенная под столбцы каждой строки, останется неосвобожденной. Поэтому сначала нужно освобождать именно память, выделенную под столбцы каждой строки, в нашем примере это «free(*(a+i))», после чего уже можно выполнить «free(a)».

На практике, конечно, использовать такую многомерную индексацию неудобно, поэтому все такие команды переписывают в привычной нотации. Ниже приведен пример абсолютно аналогичной программы, но в привычной нотации.

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#define n 3
#define m 4
int main(){
    int ** a= (int **)malloc(n*sizeof(int *));
    for (int i=0;i<n;i++)
        a[i]=(int *)malloc(m*sizeof(int));
    srand(time(NULL));
    for (int i=0;i<n;i++)
        for (int j=0;j<m;j++)
            a[i][j]=rand()%100;
    for (int i=0;i<n;i++){
        for (int j=0;j<m;j++)
            printf("%d\t",a[i][j]);
        printf("\n");
    }
    for (int i=0;i<n;i++) free(a[i]);
    free(a);
    return 0;
}
```

Также особенности возникают и при передаче многомерных массивов в функции, которые заключаются в необходимости явного указания всех размерностей массива, начиная со второй. Данное требование связано с тем, что физически любой массив представляется в памяти последовательной цепочкой байт, навигация по строкам и столбцам организуется на логическом уровне. Поэтому, если не указывать вторую и последующие размерности, останется открытым вопрос, как отсчитывать конец одной строки и начало другой. Более того, в языке С нет ограничения на перемещение по памяти с помощью указателей - даже если с помощью индексирования мы выходим за диапазон выделенной, например, под массив памяти, это не приведет к сообщению об ошибке. Контроль выхода за пределы выделенной памяти всецело возлагается на программиста.

Указатели можно использовать не только, чтобы указывать на какие-либо переменные, элементы массивов в памяти компьютера. Есть возможность ссылаться на функции.

Указатель на функцию - это переменная, содержащая адрес в памяти, по которому расположена функция, т.е. имя функции - это адрес начала программного кода функции. Указатели на функции могут быть переданы функциям в качестве аргументов, могут возвращаться функциями, сохраняться в массивах и присваиваться другим указателям на функции. Примеры использования указателей на функции будут приведены в разделе 3 настоящего учебного пособия.

Для реализации более сложных, чем динамические массивы, конструкций, в языке С используется понятие структуры - это множество логически связанных переменных, объединенных под одним именем. Такое объединение позволяет более рационально строить программы, исключить повторения кода.

Еще более интересные динамические конструкции возникают при рекурсивной ссылке некоторой структуры на саму себя. Такие структуры содержат в качестве элемента указатель, который ссылается на структуру того же типа:

```
struct node{
    int data;
    struct node * next ;
};
```

Это позволяет создавать произвольные динамические структуры данных. В качестве примера рассмотрим программу, реализующую структуру стека.

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

struct stack{
    int data;
    struct stack *next;
};
int pop(struct stack **);
void push(struct stack **, int);
int pop(struct stack **p)
{
    if ((*p) != NULL)
    {
        int a = (*p)->data;
        *p=(*p)->next;
        return a;
    }
    else
        return 0;
}
void push(struct stack **p, int element)
{
    struct stack *current;
    current = (struct stack *)
        malloc(sizeof(struct stack));
    current->data = element;
```



```

    current->next = *p;
    *p = current;
}
int main()
{
    struct stack *top_stack=NULL, *current = NULL;
    int key, a;
    do{
        printf("1_-Add, 2_-Delete; 3_-Exit\n");
        scanf("%d",&key);
        switch (key){
            case 1:
                printf("a=_"); scanf("%d",&a);
                push(&top_stack, a);
                break;
            case 2:
                printf("Element:_%d\n",pop(&top_stack));
                break;
            case 3:
                break;
            default:
                printf("Error\n");}}
    while (key != 3);
    while (top_stack != NULL)
    {
        current = top_stack->next;
        free(top_stack);
        top_stack = current;
    }
    return 0;
}

```

Пусть в стек требуется складывать приходящие числа по принципу LIFO (last input first output), а также извлекать числа по этому принципу. Для реализации структуры стека в основной

функции программы main созданы два указателя: «struct stack *top_stack» - указатель, который ссылается на вершину стека (последнее добавленное число), «struct stack *current» - вспомогательный указатель для перемещения по стеку. В main данным указателям присваивается NULL, а все действия с выделением и освобождением памяти реализованы в функциях «push» и «pop». Важно отметить, что в функции «push» и «pop» передается не сам указатель, а указатель на указатель. Это связано все с тем же обстоятельством, что по умолчанию параметры передаются в функцию по значению (т.е. передается копия значения). Однако мы должны иметь возможность менять значения указателя «*top_stack» во внешней программе main, т.е. нужно передавать параметр «*top_stack» по ссылке.

В программе также создана структура стека «struct stack», включающая в себя два поля: информационное поле «data», хранящее целое число, а также указатель, рекурсивно ссылающийся на саму структуру стека «struct stack», который хранит адрес памяти, где находится предыдущая ячейка стека.

Для доступа к элементам структуры используется две нотации: точечная и ссылочная. Если мы имеем переменную, которая сама является заданной структурой, то для доступа к полям нужно использовать точечную нотацию. Например, мы объявляем в программе переменную типа структуры стека:

```
int main(){
    struct stack a, b, *c, *d;
    a.data=1;
    b.data=2;
    a.next=&b;
    c=(struct stack *)malloc(sizeof(struct stack));
    d=(struct stack *)malloc(sizeof(struct stack));
    c->data=3; (*d).data=4;
    c->next=d; d->next=NULL;
    b.next=c;
    c=&a;
```

```

while (c<>NULL){
    printf ("%d\n",c->data);
    c=c->next;
return 0;
}

```

Здесь память под переменные «a, b» уже выделена, соответственно, мы имеем непосредственный доступ к полям структуры: «a.data=1; b.data=2; ». Чтобы связать ячейки «a, b», мы выполняем «a.next=&b» - адрес переменной «b» записываем в ссылочное поле переменной «a».

Переменные «c, d» являются лишь указателями на структуру стека, и память выделена только для хранения адреса. Например, переменным можно присвоить адреса структур: «c=&a; d=&b». Чтобы переменные «c, d» ссылались на самостоятельные структуры, нужно выделить память явным образом:

```

«c=(struct stack *)malloc(sizeof(struct stack));»,
«d=(struct stack *) malloc( sizeof (struct stack));».

```

Для доступа к элементам структуры через указатель удобно использовать ссылочную нотацию. Например, для присвоения полю «data» структуры, на которую ссылается переменная «c», значения можно использовать команду: «c->data=3;». Конечно, сохраняется и возможность доступа к полям структуры через разыменованное указателя по типу: «(*d).data», однако такой способ менее нагляден.

Как следует из приведенного фрагмента, мы сцепляем структуры «a, b, c, d» в общую структуру односвязного списка. Для перемещения по этой структуре мы присваиваем указателю «c» адрес начала цепочки (адрес переменной «a»), после чего в цикле выводим на экран значения информационных частей переменных «a, b, c, d», перемещаясь по структуре с помощью изменения адреса указателя «c=c->next;».

2.4 Вопросы для самоконтроля

Источники: [6].

1. Что называется машинно-зависимым языком программирования? Какие языки машинно-зависимы?
2. Назовите два класса языков программирования высокого уровня и их основные отличия.
3. Приведите примеры компилируемых и интерпретируемых языков программирования.
4. Дайте краткую характеристику каждой основной стадии разработки программы на компилируемом языке высокого уровня.
5. Кем был разработан язык программирования C? Кем была написана книга «Программирование на языке C»?
6. Что такое ANSI C?
7. Установите, являются ли следующие утверждения верными:
 - Когда вызывается функция `printf`, она всегда начинает печать с начала новой строки.
 - Все переменные должны быть объявлены, прежде чем будут использоваться.
 - Язык C рассматривает переменные `number1` и `Number1` как тождественные.
 - Операция взятия модуля `%` может использоваться только с целыми операндами.
 - Арифметические операции `*`, `/`, `%` имеют одинаковый приоритет.
 - Чтобы вывести на экран три строки, необходимо использовать три оператора `printf`.

8. Найдите и исправьте ошибки:

```
scanf("d", value);
```

```
printf("The_sum_is_%d\n", x+y);
```

```
*/ My program /*
```

9. Определите, какое значение получит переменная x :

- $x = 7 + 3 * 6 / 2 - 1$;
- $x = 2 \% 2 + 2 * 2 - 2 / 2$;
- $x = (3 * 9 * (3 + (9 * 3 / (3))))$;

10. Напишите программу, которая предлагает пользователю ввести два целых числа, получает эти числа и после этого выводит на печать большее из чисел со словами «это большее». Если же числа равны, должно печататься сообщение «Эти числа равны».

11. Напишите одиночный оператор C для выполнения каждой из следующих задач:

- Присвойте сумму x и y переменной z и увеличьте значение переменной x на 1 после выполнения вычислений.
- Уменьшите значение переменной x на 1, затем вычтите его из переменной $total$.
- Выведите значение 123.4567 с точностью до 2 знаков. Какое значение будет выведено?

12. Определите значение каждой из переменных после выполнения вычисления при $x=5$, $product = 5$:

```
product *= x++;
```

```
result = ++x + x;
```

13. Исправьте ошибки:

```
while (c <= 5);  
{  
product *= c;  
++c
```

```
scanf("%f", &v);
```

```
if (g = 1)  
    printf("Woman\n");  
else;  
    printf("Man\n");
```

14. Что неправильно в следующей структуре повторения while

```
sum=0; z = 0;  
while ( z >= 0)  
sum += z;
```

15. Установите, являются ли следующие утверждения верными или ложными. Если утверждение ложно, объясните почему.

- В структуре выбора **switch** необходим блок **default**.
- В блоке **default** структуры **switch** необходим оператор **break**.
- Выражение $(x > y \ \&\& \ a < b)$ истинно, если истинно либо выражение $x > y$, либо выражение $a < b$.
- Выражение, содержащее операцию **||**, истинно, если истинен один или оба операнда.

16. Напишите оператор **C** или последовательность операторов для решения каждой из следующих задач:

- Просуммируйте нечетные числа от 1 до 99, используя для этого структуру **for**.
- Выведите значение 3.141592653589793 с шириной поля 20 символов с точностью 1, 2, 3, 4, 5 знаков после точки. Выровняйте выводимые значения по левому краю.
- Вычислите значение 2.5 в степени 3, используя для этого функцию **pow**. Выведите результат с точностью 2 и шириной поля 10 символов.

17. Найдите ошибки в блоке программного кода

Код должен выводить значения от 1 до 10

```
n=1;
while (n<10)
    printf(" %d", n++);
```

18. Найдите ошибки в следующих фрагментах кода:

```
for (i = 100, x >= 1, x++)
    printf(" %d\n", x);
```

19. Дайте ответ на следующие вопросы.

- Как называется модуль программы на C?
- Посредством чего осуществляется обращение к функции?
- Как называется переменная, которая известна только внутри функции, в которой она определена?
- Как называется оператор, который используется в вызываемой функции для передачи значения вызывающей функции?
- Как называется ключевое слово, которое используется в заголовке функции, чтобы показать, что функция не возвращает значения или не содержит никаких параметров?

20. Дайте ответ на следующие вопросы.

- Как называется часть программы, в которой идентификатор может быть использован?
- Назовите три способа возврата из функции.
- Какой механизм позволяет компилятору проверять количество аргументов, типы аргументов, порядок их следования, а также тип возвращаемого значения функции?
- Какая функция используется для генерации случайных чисел?
- Какая функция выбирает последовательность генератора случайных чисел?

21. Дайте ответ на следующие вопросы.

- Назовите 4 спецификатора класса памяти.
- К какому спецификатору класса памяти относится переменная, объявленная в блоке или списке параметров функции, если нет никаких специальных указаний?
- Какой спецификатор класса памяти рекомендует компилятору разместить переменную в одном из регистров процессора?

22. Дайте ответ на следующие вопросы.

- С каким спецификатором класса памяти должна быть объявлена локальная переменная, чтобы она сохраняла свое значение между вызовами функции?
- Назовите четыре возможные области действия идентификатора.
- Как называется функция, которая вызывает саму себя непосредственно или косвенно?

23. Заполните пропуски в предложениях.

- Списки и таблицы значений можно хранить в
- Элементы массива связаны в том отношении, что они имеют одинаковое, а также
- Число, используемое для обращения к конкретному элементу массива, называется
- Для объявления размера массива целесообразно использовать, поскольку в этом случае программа становится более общей.
- Процесс размещения элементов массива в определенном порядке называется массива.

24. Заполните пропуски в предложениях.

- Процесс определения, содержит ли массив некоторое ключевое значение, называется в массиве
- Массив, который имеет два индекса, называется
- При ссылке на элемент массива номер его позиции указывается в скобках
- В функцию массив целесообразно передавать вместе с его
- В прототипе функции массив целых чисел указывается

25. Установите, являются ли верными следующие утверждения.

- В массиве может храниться много различных типов значений.
- Индекс массива может быть числом типа float.
- Если число инициализирующих значений меньше числа элементов массива, оставшиеся элементы автоматически инициализируются последним значением в списке инициализации.

- Если список инициализации содержит больше инициализирующих значений, чем имеется элементов в массиве, это является ошибкой.
- Отдельный элемент массива, который передается в функцию и изменяется в ней, изменится и в самом массиве.

26. Найдите ошибку во фрагментах программы:

```
#define SIZE 100;
```

```
A[1,1]=5;
```

```
int A[5], i;  
for (i = 0; i <= 5; i++) A[i] = i;
```

```
#include <stdio.h>;  
void f(int A[10]);  
void g(int B[][]);
```

27. Напишите операторы для выполнения каждой из следующих задач:

- Отобразите на экране значение седьмого элемента символьного массива f.
- Просуммируйте все элементы массива c, состоящего из 100 элементов с плавающей точкой.
- Скопируйте массив a[11] в начало массива b[20].
- Скопируйте массив a[11] в конец массива b[20].
- Определите и выведите самое большое и самое маленькое число в массиве A[99], состоящего из элементов с плавающей точкой.

- Инициализируйте каждый элемент массива `s[10]` числом 999.

28. Заполните пропуски в предложениях.

- Указатель - это переменная, которая в качестве значения содержит другой переменной.
- Только три величины могут использоваться для инициализации указателя, или
- Единственное число, которое может быть присвоено указателю, - это

29. Являются ли следующие утверждения верными? Если утверждение неверно, объясните почему.

- Операция взятия адреса `&` может применяться только к константам, выражениям и переменным, объявленным с модификатором `register`.
- Указатель на `void` может быть разыменован.
- Указатели на разные типы данных не могут быть присвоены друг другу без использования операции приведения типов.

3.1 Суммирование рядов и вычисление элементарных функций

3.1.1 Пример вычисления суммы ряда

Продemonстрируем разработку численных алгоритмов на примере различных методов вычисления числа π . Начнем с известного своей медленной сходимостью ряда Грегори-Лейбница:

$$\pi = 4 \cdot \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots \right).$$

Разработку алгоритма начнем с блок-схемы, представленной на рис. 3.1.

По данной блок-схеме легко перейти к разработке программного кода, листинг которого представлен ниже.

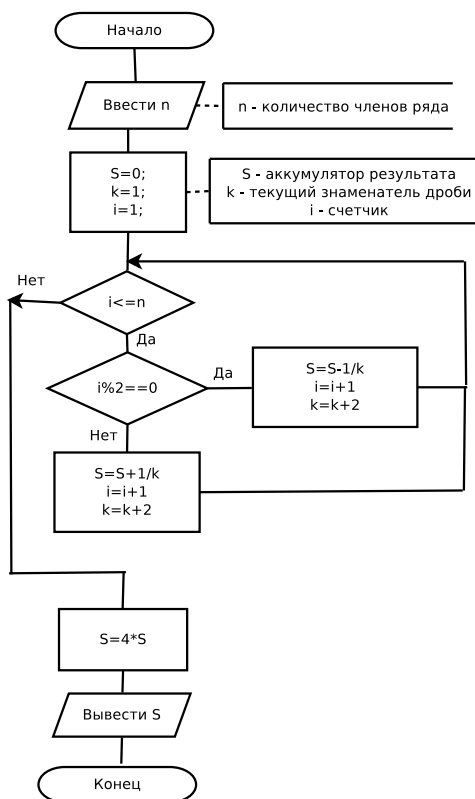


Рис. 3.1. Блок-схема алгоритма вычисления ряда Грегори-Лейбница

Из листинга видно, что, кроме алгоритма вычисления суммы ряда, в программном коде реализовано сравнение полученного значения с точным значением числа π , взятым из библиотеки «math».

Для приемлемой точности, например 6 верных знаков после плавающей точки, потребуется достаточно большое число членов ряда.

На рис. 3.2 показаны сравнительные расчеты при разном количестве членов ряда. Видно, что увеличение количества членов ряда в 1000 раз приводит в среднем к увеличению точности всего на три верных знака.

```
#include <stdio.h>
#include<math.h>
int main(){
    unsigned long long int n,k=1,i;
    long double S=0;
    printf("n=");
    scanf("%llu", &n);
    for (i=1;i<=n;i++){
        if (i%2)
            {S+=(long double)1/k; k+=2;}
        else {S-=(long double)1/k; k+=2;}
    }
    S*=4;
    printf("For_%llu_elements_S=%.20Lf\n",n,S);
    printf("The_absolute_error:_%%.20Lf\n",M_PI-S);
    return 0;
}
```

Для оптимизации вычислений часто от знакопеременного ряда переходят к ряду из положительных членов, выполняя попарное сложение членов знакопеременного ряда:

$$\left(1 - \frac{1}{3}\right) + \left(\frac{1}{5} - \frac{1}{7}\right) + \dots = \sum_{i=0}^n \left(\frac{1}{4i+1} - \frac{1}{4i+3}\right),$$

$$\sum_{i=0}^n \left(\frac{1}{4i+1} - \frac{1}{4i+3}\right) = \sum_{i=0}^n \left(\frac{2}{(4i+1)(4i+3)}\right).$$

```
juna@artamonov: ~/Документы/TeXSubject/Programming/Cursov/Code
Файл Правка Вид Поиск Терминал Справка
juna@artamonov:~/Документы/TeXSubject/Programming/Cursov/Code$ ./a.out
Введите число членов ряда n=1000
Сумма 1000 членов ряда Грегори-Лейбница составила 3.14059265383979292654
Абсолютная погрешность составила: 0.00099999975000018945
juna@artamonov:~/Документы/TeXSubject/Programming/Cursov/Code$ ./a.out
Введите число членов ряда n=1000000
Сумма 1000000 членов ряда Грегори-Лейбница составила 3.14159165358979318525
Абсолютная погрешность составила: 0.00000099999999993074
juna@artamonov:~/Документы/TeXSubject/Programming/Cursov/Code$ ./a.out
Введите число членов ряда n=1000000000
Сумма 1000000000 членов ряда Грегори-Лейбница составила 3.14159265258979522790
Абсолютная погрешность составила: 0.0000000099999788810
juna@artamonov:~/Документы/TeXSubject/Programming/Cursov/Code$
```

Рис. 3.2. Результаты вычислений при разном количестве членов ряда

Таким образом, можно использовать представление:

$$\pi = 4 \cdot \sum_{i=0}^n \left(\frac{2}{(4i+1)(4i+3)} \right).$$

Ниже представлен листинг программного кода, реализующего данный способ вычислений.

```
#include <stdio.h>
#include<math.h>
int main(){
    unsigned long long int n,k=1,i;
    long double S=0;
    printf("n=");
    scanf("%llu", &n);
    for (i=0;i<=n-1;i++)
        S+=(long double)2/((4*i+1)*(4*i+3));
    S*=4;
    printf("For_%llu_elements_S=%.20Lf\n",n,S);
    printf("The_absolute_error:_.%.20Lf\n",M_PI-S);
    return 0; }
```

Однако, как видно из рис. 3.3, добиться существенного улучшения точности при таком методе расчета все равно не получается.

```

juna@artamonov: ~/Документы/TeXSubject/Programming/Cursov/Code
Файл Правка Вид Поиск Терминал Справка
juna@artamonov:~/Документы/TeXSubject/Programming/Cursov/Code$ ./a.out
Введите число членов ряда n=1000
Сумма 1000 членов ряда Грегори-Лейбница составила 3.14109265362104322829
Абсолютная погрешность составила: 0.00049999996874988770
juna@artamonov:~/Документы/TeXSubject/Programming/Cursov/Code$ ./a.out
Введите число членов ряда n=1000000
Сумма 1000000 членов ряда Грегори-Лейбница составила 3.14159215358979327227
Абсолютная погрешность составила: 0.000004999999984372
juna@artamonov:~/Документы/TeXSubject/Programming/Cursov/Code$ ./a.out
Введите число членов ряда n=1000000000
Сумма 1000000000 членов ряда Грегори-Лейбница составила 3.14159265308978679688
Абсолютная погрешность составила: 0.0000000050000631912
juna@artamonov:~/Документы/TeXSubject/Programming/Cursov/Code$

```

Рис. 3.3. Результаты вычислений при разном количестве членов ряда для модифицированного ряда

Леонард Эйлер придумал оригинальный способ использования ряда Лейбница-Грегори для вычисления числа π с достаточной точностью при малом числе членов ряда [7]. Во-первых, нужно отметить, что ряд Лейбница-Грегори получается из более общего разложения для арктангенса угла:

$$\arctg(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \dots$$

Положив $x = 1$, получаем ряд Лейбница-Грегори. Далее используем известное всем соотношение:

$$\operatorname{tg}(\alpha + \beta) = \frac{\operatorname{tg}(\alpha) + \operatorname{tg}(\beta)}{1 - \operatorname{tg}(\alpha) \operatorname{tg}(\beta)}.$$

Л. Эйлер взял: $\alpha = \arctg\left(\frac{1}{2}\right)$, $\beta = \arctg\left(\frac{1}{3}\right)$, тогда получится:

$$\operatorname{tg}(\alpha + \beta) = \frac{\frac{1}{2} + \frac{1}{3}}{1 - \frac{1}{2} \cdot \frac{1}{3}} = 1 \Rightarrow \arctg\left(\frac{1}{2}\right) + \arctg\left(\frac{1}{3}\right) = \arctg(1) = \frac{\pi}{4},$$

$$\operatorname{arctg}\left(\frac{1}{2}\right) = \frac{1}{2} - \frac{1}{2^3 \cdot 3} + \frac{1}{2^5 \cdot 5} - \frac{1}{2^7 \cdot 7} + \dots,$$

$$\operatorname{arctg}\left(\frac{1}{3}\right) = \frac{1}{3} - \frac{1}{3^3 \cdot 3} + \frac{1}{3^5 \cdot 5} - \frac{1}{3^7 \cdot 7} + \dots$$

Окончательно приходим к такому представлению:

$$\begin{aligned} \pi = & 4 \cdot \left(\frac{1}{2} - \frac{1}{2^3 \cdot 3} + \frac{1}{2^5 \cdot 5} - \frac{1}{2^7 \cdot 7} + \dots \right) + \\ & 4 \cdot \left(\frac{1}{3} - \frac{1}{3^3 \cdot 3} + \frac{1}{3^5 \cdot 5} - \frac{1}{3^7 \cdot 7} + \dots \right). \end{aligned}$$

Для реализации данного представления удобно в программном коде использовать вычисление функции арктангенса от числа x . Ниже представлен листинг данной программы.

```
#include <stdio.h>
#include<math.h>
long double atg(long double,
                unsigned long long int);
long double atg(long double x,
                unsigned long long int n){
    long double S=0, xx=x;
    unsigned long long int k=1,i;
    int sign=1;
    for (i=1;i<=n;i++){
        S+=sign*xx/k;
        k+=2;
        sign*=-1;
        xx*=x*x;
    }
    return S;
}

int main(){
```

```

unsigned long long int n;
printf("n=");
scanf("%llu", &n);
long double r;
r=4*atg((long double)1/2,n)+
  4*atg((long double)1/3,n);
printf("For_%llu_elements_S=%.20Lf\n",n,r);
printf("The_absolute_error: %.20Lf\n",M_PI-r);
return 0;
}

```

```

juna@artamonov: ~/Документы/TeXSubject/Programming/Curosov/Code
Файл Правка Вид Поиск Терминал Справка
juna@artamonov:~/Документы/TeXSubject/Programming/Curosov/Code$ ./a.out
n=10
For 10 elements S= 3.14159257960635121109
The absolute error: 0.00000007398344190491
juna@artamonov:~/Документы/TeXSubject/Programming/Curosov/Code$ ./a.out
n=100
For 100 elements S= 3.14159265358979323830
The absolute error: -0.00000000000000012230
juna@artamonov:~/Документы/TeXSubject/Programming/Curosov/Code$

```

Рис. 3.4. Результаты вычислений при разном количестве членов ряда в модификации Л. Эйлера

На рис. 3.4 представлены результаты вычислений, основанные на идее Л. Эйлера. Как видно из рисунка, уже при 10 членах ряда получаем точность в 7 верных знаков, а при 100 членах ряда получаем 15 верных знаков, которые мы не могли получить в ряде Грегори-Лейбница даже на миллиарде членов ряда.

3.1.2 Задания для самостоятельной работы

Источники: [2, 10, 11].

1. Для функции $\operatorname{tg}(x)$ имеет место представление в виде цепной дроби:

$$\operatorname{tg}(x) = \frac{1}{\frac{1}{x} - \frac{1}{3 - \frac{1}{x - \frac{1}{5 - \frac{1}{x - \frac{1}{7 - \frac{1}{x - \dots}}}}}}}}.$$

Реализуйте функцию, вычисляющую $\operatorname{tg}(x)$ по этому представлению, и проверьте, насколько быстро сходится процесс вычислений (т.е. сколько членов дроби надо взять для получения результата с заданной точностью при различных значениях x). Полученные результаты расчета оформите в виде таблицы в пояснительной записке: значение x , точность, количество членов дроби.

2. Известны два представления числа e в виде ряда и бесконечной дроби:

$$e = 2 + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \dots,$$

$$e = 2 + \frac{2}{2 + \frac{3}{3 + \frac{4}{4 + \dots}}}.$$

Сравните скорости сходимости с помощью ряда и дроби. Полученные результаты расчета оформите в виде таблицы в пояснительной записке: точность, количество членов ряда, количество членов дроби.

3. Известны два представления числа π :

$$\pi = 3 + 4 \cdot \left(\frac{1}{2 \cdot 3 \cdot 4} - \frac{1}{4 \cdot 5 \cdot 6} + \frac{1}{6 \cdot 7 \cdot 8} - \dots \right),$$
$$\pi = \sqrt{6 \cdot \left(1 + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \dots \right)}.$$

Сравните скорость сходимости каждого представления. Полученные результаты расчета оформите в виде таблицы в пояснительной записке: точность, количество членов ряда.

4. Известно следующее представление:

$$\prod_{k=1}^{\infty} \left(1 + \frac{(-1)^k}{2k+1} \right) = \frac{\sqrt{2}}{2}.$$

Определите, сколько сомножителей потребуется взять, чтобы равенство выполнялось до шестой значащей цифры. Полученные результаты расчета оформите в виде таблицы в пояснительной записке: точность, количество членов ряда.

5. Известно следующее представление:

$$\frac{\pi^2}{8} - \frac{\pi}{4}|x| = \frac{\cos(3x)}{3^2} + \frac{\cos(5x)}{5^2} + \dots + \frac{\cos((2n+1)x)}{(2n+1)^2} + \dots,$$
$$|x| < 1.$$

Реализуйте вычисление по этому представлению и проверьте, насколько быстро сходится процесс вычислений (т.е. сколько слагаемых надо взять для получения результата с заданной точностью при различных значениях x). Полученные результаты расчета оформите в виде таблицы в пояснительной записке: значение x , точность, количество членов ряда.

6. Известно следующее представление:

$$\frac{1}{4} \left(x^2 - \frac{\pi^2}{3} \right) = -\cos(x) + \frac{\cos(2x)}{2^2} - \dots + (-1)^n \frac{\cos(nx)}{n^2},$$

$$\frac{\pi}{5} \leq x \leq \pi.$$

Реализуйте вычисление по этому представлению и проверьте, насколько быстро сходится процесс вычислений (т.е. сколько слагаемых надо взять для получения результата с заданной точностью при различных значениях x). Полученные результаты расчета оформите в виде таблицы в пояснительной записке: значение x , точность, количество членов ряда.

7. Известно следующее представление:

$$\frac{1}{4} \ln \left(\frac{1+x}{1-x} \right) + \frac{1}{2} \operatorname{arctg}(x) = x + \frac{x^5}{5} + \dots + \frac{x^{4n+1}}{4n+1} + \dots,$$

$$-1 < x < 1.$$

Реализуйте вычисление по этому представлению и проверьте, насколько быстро сходится процесс вычислений (т.е. сколько слагаемых надо взять для получения результата с заданной точностью при различных значениях x). Полученные результаты расчета оформите в виде таблицы в пояснительной записке: значение x , точность, количество членов ряда.

8. Известно следующее представление:

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

Реализуйте вычисление по этому представлению и проверьте, насколько быстро сходится процесс вычислений

(т.е. сколько слагаемых надо взять для получения результата с заданной точностью при различных значениях x). Полученные результаты расчета оформите в виде таблицы в пояснительной записке: значение x , точность, количество членов ряда.

9. Известно следующее замечательное соотношение, установленное С.Рамануджаном:

$$\sqrt{\frac{e \cdot \pi}{2}} = 1 + \frac{1}{1 \cdot 3} + \frac{1}{1 \cdot 3 \cdot 5} + \frac{1}{1 \cdot 3 \cdot 5 \cdot 7} + \dots +$$

$$+ \frac{1}{1 + \frac{1}{1 + \frac{2}{1 + \frac{3}{1 + \frac{4}{1 + \dots}}}}}.$$

Вычислить, сколько членов ряда и цепной дроби нужно взять, чтобы достичь заданной точности. Полученные результаты расчета оформите в виде таблицы в пояснительной записке: значение x , точность, количество членов ряда, количество членов дроби.

10. Известно представление $\cos(x)$ в виде произведения:

$$\cos(x) = \left(1 - \frac{4x^2}{\pi^2}\right) \left(1 - \frac{4x^2}{9\pi^2}\right) \dots \left(1 - \frac{4x^2}{(2n-1)^2\pi^2}\right).$$

Реализуйте функцию, вычисляющую $\cos(x)$ по этому представлению и проверьте, насколько быстро сходится процесс вычислений (т.е. сколько множителей надо взять для получения результата с заданной точностью при различных значениях x). Полученные результаты расчета оформите в виде таблицы в пояснительной записке: значение x , точность, количество множителей.

Распределение заданий по вариантам

Номер варианта:	1	2	3	4	5	6	7	8	9	10	11
Номер задания:	5	9	1	2	7	8	7	3	4	6	1

Номер варианта:	12	13	14	15	16	17	18	19	20
Номер задания:	2	3	4	5	6	7	8	9	10

Номер варианта:	21	22	23	24	25	26	27	28	29
Номер задания:	9	3	4	5	1	7	8	2	10

3.2 Приближенные методы нахождения корней уравнения

3.2.1 Пример использования метода последовательных приближений

Вычислительные методы широко используются и для нахождения корней различных математических уравнений.

Пусть требуется найти значение x_0 , при котором для заданной функции $f(x)$ получится $f(x_0) = 0$. Одним из методов решения подобной задачи является *метод последовательных приближений* [3]. Суть метода состоит в следующем. От уравнения $f(x) = 0$ переходят к эквивалентному уравнению

$$\varphi(x) = x.$$

Далее, задавшись начальным приближением $x = x_1$, вычисляют:

$$x_2 = \varphi(x_1),$$

считая полученное значение x_2 вторым приближением.

Повторяя подобные преобразования n раз, находят n -е приближение корня уравнения $f(x)$:

$$x_n = \varphi(x_{n-1}).$$

Продemonстрируем использование данного метода на конкретном примере. Пусть требуется найти корень уравнения

$$2x + \cos(x) - \ln(x) = 0.$$

В нашем случае $f(x) = 2x + \cos(x) - 10 \ln(x)$, приводим уравнение к виду $\varphi(x) = x$:

$$x = \frac{10 \ln(x) - \cos(x)}{2}.$$

Зададимся начальным приближением $x_1 = 13$, тогда получим следующую цепочку вычислений:

$$x_2 = \varphi(13) \approx 12.37102339658258,$$

$$x_3 = \varphi(x_2) \approx 12.08629441000185,$$

$$x_4 = \varphi(x_3) \approx 12.0168807208529,$$

...

$$x_{10} \approx 12.00306265921456,$$

...

$$x_{20} \approx 12.00306250123322.$$

Дальнейшее увеличение количества итераций приводит к стабилизации. Найденное решение действительно является корнем рассматриваемого уравнения:

$$2(12.003) + \cos(12.003) - 10 \ln(12.003) \approx -0.0000106.$$

В качестве обоснования для данного метода можно рассмотреть совмещенные графики функций $y = x, \varphi(x) = \frac{10 \ln(x) - \cos(x)}{2}$.

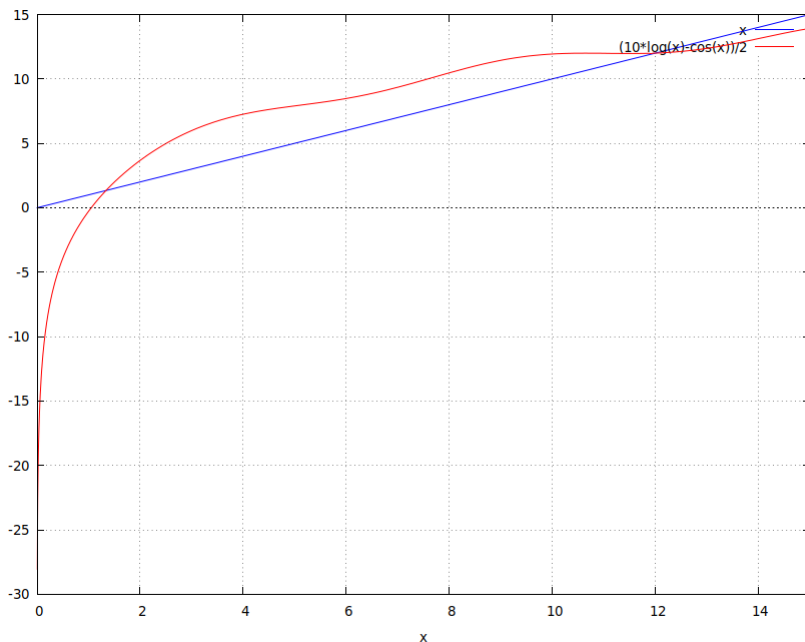


Рис. 3.5. Совмещение графиков функций $\varphi(x)$, $y = x$

Как видно из рис. 3.5, данные графики действительно пересекаются в точке $x \approx 12$.

На рис. 3.6 показан увеличенный фрагмент графика в диапазоне значений $x \in [11, 14]$, а также продемонстрирована идея метода последовательных приближений: фактически осуществляется своеобразный последовательный спуск по графикам $\varphi(x)$ и $y = x$ к точке их пересечения, начиная с начального приближения $x_1 = 13$.

Возникает вопрос, всегда ли метод последовательных приближений позволяет получить один из корней любого уравнения $f(x) = 0$, которое можно привести к виду $x = \varphi(x)$?

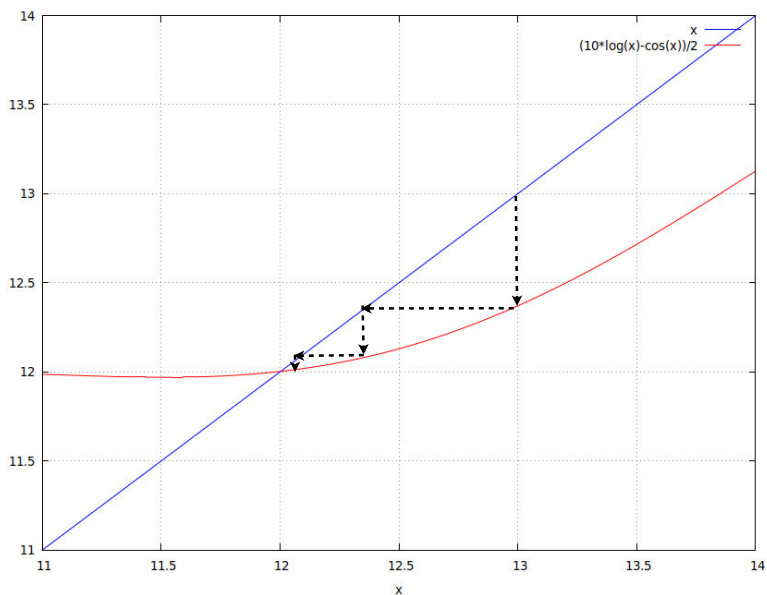


Рис. 3.6. Наглядная демонстрация идеи метода последовательных приближений

Ответ на этот вопрос отрицательный, точный критерий устанавливает следующая теорема [3].

Метод последовательных приближений сходится к корню уравнения $x = \varphi(x)$ на отрезке $[a, b]$, если для всех точек этого отрезка $\xi \in [a, b]$ имеем $|\varphi'(\xi)| < 1$.

Так, в рассматриваемом примере имеем:

$$\varphi'(x) = \frac{5}{x} + \frac{1}{2} \sin(x).$$

На рис. 3.7 показан график $\varphi'(x)$, видно, что на отрезке $[10, 14]$ условие $|\varphi'(\xi)| < 1$ соблюдается в каждой точке этого отрезка,

поэтому метод последовательных приближений сходится к одному из корней.

В то же время на отрезке $[1, 3]$, где располагается второй корень, условие теоремы не выполнено, поэтому данный корень не может быть найден методом последовательных приближений.

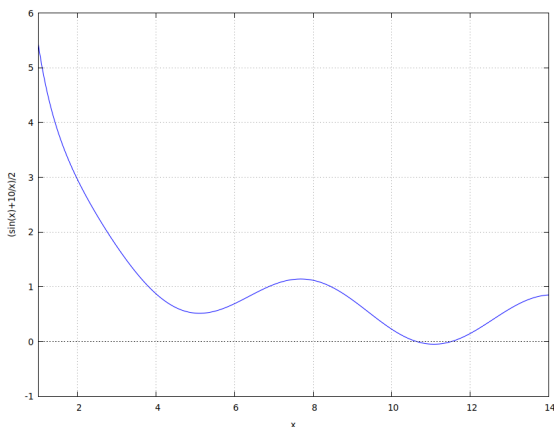


Рис. 3.7. Анализ сходимости метода последовательных приближений

Заметим также, что процесс получения из функции $f(x)$ функции $\varphi(x)$ неоднозначен. Например, в нашем случае при $f(x) = 2x + \cos(x) - 10 \ln(x)$ можно было выполнить такие преобразования:

$$\ln(x) = \frac{2x + \cos(x)}{10} \Rightarrow x = e^{\frac{2x + \cos(x)}{10}} \Rightarrow \varphi(x) = e^{\frac{2x + \cos(x)}{10}}.$$

Причем, как можно заметить, в этом случае:

$$\varphi'(x) = \frac{e^{\frac{2x + \cos(x)}{10}} \cdot (2 - \sin(x))}{10}, |\varphi'(\xi)| < 1, \xi \in [1, 3].$$

Действительно, как видно из рис. 3.8, в диапазоне $[1, 3]$ получаем $|\varphi'(x)| < 1$.

Это позволяет найти второй корень уравнения $f(x) = 0$ на отрезке $[1, 3]$, показанный на рис. 3.5.

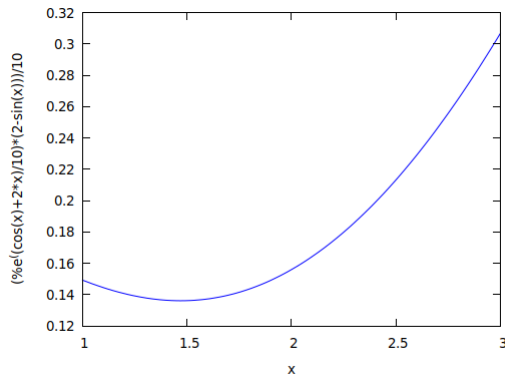


Рис. 3.8. Анализ сходимости при модификации функции $\varphi(x)$

Зададимся начальным приближением $x_1 = 1.5$, далее для $\varphi(x) = e^{\frac{2x + \cos(x)}{10}}$ получаем:

$$x_2 = \varphi(1.5) \approx 1.359441182608715,$$

$$x_3 = \varphi(x_2) \approx 1.340264181113018,$$

$$x_4 = \varphi(x_3) \approx 1.337634022601982,$$

...

$$x_{10} \approx 1.337215334836982,$$

...

$$x_{20} \approx 1.337215332025681.$$

Проверяем полученное решение:

$$2 \cdot 1.33721533 + \cos(1.33721533) - 10 \cdot \ln(1.33721533) \approx 1.307 \cdot 10^{-8}.$$

Для построения графиков функций, нахождения производных, анализа поведения функции на заданном отрезке удобно использовать систему символьных вычислений *maxima*. *Maxima* является свободным программным обеспечением с лицензией GNU GPL. Скачать ее можно с официального сайта <https://maxima.sourceforge.io/>.

В операционных системах семейства GNU Linux (дистрибутив Debian или Ubuntu) для установки *maxima* и рабочего окружения с ней (*emacs*) можно использовать следующую команду:

```
>sudo apt-get install emacs maxima maxima-emacs
```

После установки указанных пакетов следует запустить *emacs*, нажать комбинацию клавиш **Alt+x** и выполнить одну из команд:

```
>maxima  
>imaxima
```

Первая команда запускает *maxima* в обычном текстовом режиме, такой режим удобно использовать, если в дальнейшем требуется копировать результаты вычислений в виде текста. Вторая команда запускает *maxima* в режиме интерпретатора системы \TeX , создавая на выходе графические объекты для наглядной визуализации математических выражений.

В качестве примера на рис. 3.9 показан скриншот экранной формы системы *maxima*, запущенной в режиме интерпретатора \TeX , где задается функция $\varphi(x) = \frac{10 \ln(x) - \cos(x)}{2}$ и строится ее график, а также график функции $y = x$ на отрезке $[0, 15]$.

Как видно из рис. 3.9, получился результат, показанный на рис. 3.5. Причем на графике возможна навигация, выделение областей, получение координат выделенных точек.

Также с легкостью в системе *maxima* можно получить и построить график производной функции, например, с помощью команд:

```
> g(x):=diff(phi(x),x); plot2d(g(x),[x,1,15]);
```

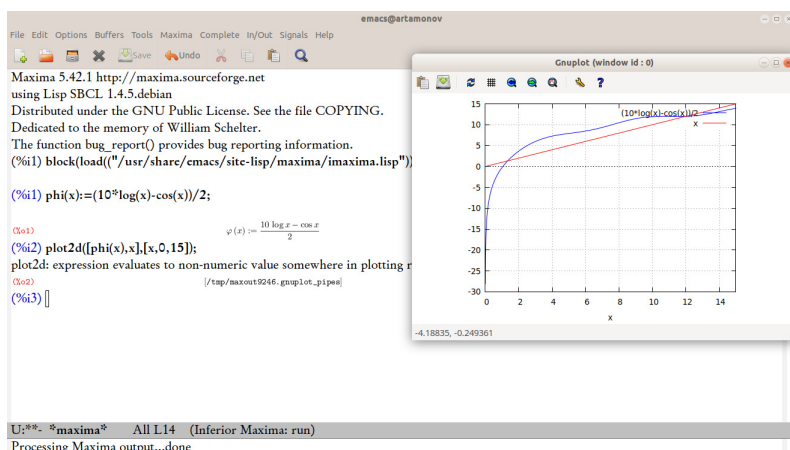


Рис. 3.9. Пример работы в системе maxima

Результат выполнения данных команд показан на рис. 3.10. Как видно из рисунка, график совпадает с изображением на рис. 3.7.

При написании программного кода удобно декомпозировать задачу на ряд отдельных задач:

- блок 1 вычисления функций $\varphi(x)$, а также их производных $\varphi'(x)$;
- блок 2 реализации метода последовательных приближений;
- блок 3 интерфейса с пользователем.

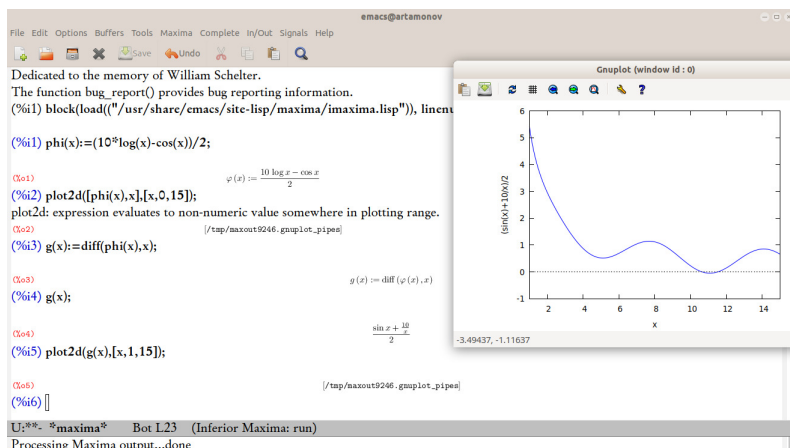


Рис. 3.10. Пример работы в системе maxima - получение производной функции

Блок 1 реализуем в виде библиотеки. В качестве функций возьмем рассмотренные выше:

$$\varphi_1(x) = \frac{10 \ln(x) - \cos(x)}{2}, \varphi_1'(x) = \frac{5}{x} + \frac{1}{2} \sin(x),$$

$$\varphi_2(x) = e^{\frac{2x + \cos(x)}{10}}, \varphi_2'(x) = \frac{e^{\frac{2x + \cos(x)}{10}} \cdot (2 - \sin(x))}{10}.$$

Ниже представлен листинг соответствующего программного кода, сохраненного в файле с именем func.c.

```

#include<math.h>
double phi1(double x){
    return (10*log(x)-cos(x))/2;
}
double dphi1(double x){
    return 5/x+1/(2*sin(x));
}
  
```

```

}
double phi2(double x){
    return exp((2*x+cos(x))/10);
}
double dphi2(double x){
    return exp((2*x+cos(x))/10)*(2 - sin(x))/10;
}

```

Блок 2 также оформим отдельной библиотекой, которую потом подключим в основной программе. Поскольку алгоритм работы модуля не зависит от того, для какой функции и ее производной проводятся вычисления, его целесообразно реализовать с использованием указателей на функции. Таким образом, определим функцию с именем «method», которая принимает на вход следующий состав параметров: указатель на функцию φ , указатель на производную функции $\varphi'(x)$, начальное значение для метода последовательных приближений, изменяемая переменная - признак сходимости процесса приближений. Ниже показан листинг соответствующего программного кода, сохраненного с именем «method.c».

```

double method(double (*f)(double),
               double (*df)(double),
               double val,
               int n,
               int *p)
{
    double xx_new=val, xx_old=val, current;
    int pr=1,i=1;
    while ((i<=n) && pr)
    {
        current=xx_new;
        xx_new=f(xx_old);
        if (fabs(df(xx_old))>=1) pr=0;
        xx_old=current;
    }
}

```



```
        i++;  
    }  
    *p=pr;  
    return xx_new;  
}
```

Существует два основных способа использования созданных библиотечных файлов: построение *статических и динамических библиотек*.

Динамические библиотеки хранятся отдельно от исполняемых файлов, как правило, в системных директориях. Они могут использоваться сразу многими исполняемыми файлами. Например, библиотека «math» языка С. Еще одно преимущество этого способа - сравнительно малый объем исполняемого файла. Однако в таком способе есть и недостатки. Если у пользователя нет в системе соответствующей динамической библиотеки, исполняемый файл работать не будет.

Процесс сборки динамической библиотеки включает в себя следующие этапы:

- Компиляция библиотечных файлов без линковки (для gcc это использование ключа компиляции -с). В нашем случае этот процесс будет выглядеть так:

```
> gcc -c func.c
```

```
> gcc -c method.c
```

В результате выполнения этих команд в текущей директории будет создано два объектных файла: func.o, method.o

- Создание динамической библиотеки из объектных файлов. В gcc для этих целей необходимо использовать ключ компиляции -shared, указывая имя библиотеки с расширением .so, а также имена объектных файлов, из которых она будет построена. В нашем случае этот процесс будет выглядеть так:

```
> gcc -shared -o libmy.so func.o method.o
```

После этого в текущей директории будет создана динамическая библиотека с именем `libmy.so`.

- Чтобы получить доступ к динамической библиотеке, исполняемый файл должен знать, где она находится. Для пользовательских библиотек в системе GNU Linux чаще всего используется системная директория `/usr/lib`. Именно туда и нужно переместить созданный файл `libmy.so`.

```
> sudo mv ./libmy.so /usr/lib
```

Обратите внимание, поскольку директория `/usr/lib` системная, требуются права администратора (команда `sudo`).

Вместо физического перемещения файла `libmy.so` можно создать на него символическую ссылку:

```
> sudo ln -s ./libmy.so /usr/lib/
```

Для использования разработанного функционала создадим заголовочный файл, в котором опишем прототипы функций из динамической библиотеки. Заметим, что в самой динамической библиотеке функций может быть намного больше, чем тех, что мы указываем в заголовочном файле. Заголовочный файл и служит для перечисления только требуемых функций.

Листинг соответствующего заголовочного файла, сохраненного с именем «`my.h`», представлен ниже.

```
double phi1(double);
double phi2(double);
double dphi1(double);
double dphi2(double);
double method(double (*)(double),
               double (*)(double),
               double, int, int *);
```

Теперь остается разработать интерфейсный модуль (блок 3), в котором будет вызываться и использоваться весь разработанный функционал.

Данный интерфейсный модуль содержит функцию `main`. Для удобства работы с этим модулем предусмотрим передачу параметров исполняемому файлу из командной строки. С этой целью для функции `main` используется специальная форма:

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

Здесь целочисленная переменная `argc` хранит количество параметров, переданных исполняемой программе из командной строки, `**argv` хранит указатель на начало строки параметров (принято соглашение, что каждый параметр отделяется от другого не менее чем одним пробельным символом). Для доступа к соответствующим параметрам используется индексация `*argv`. При этом `*argv[0]` хранит имя исполняемого файла, `*argv[1]` хранит в виде строки значение первого параметра, `*argv[2]` хранит в виде строки значение второго параметра и т.д.

В нашем случае будем передавать параметры в следующем порядке: первый аргумент - имя функции $\varphi(x)$, второй аргумент - имя функции $\varphi'(x)$, третий аргумент - начальное приближение, четвертый аргумент - количество итераций.

Ниже представлен листинг программного кода для блока 3, сохраненного с именем `main.c`.

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

int main(int argc , char **argv){
    int p=1,n;
    double val;
    double (*f)(double), (*df)(double);
    if (!(strcmp(argv[1],"phi1"))){ f=phi1;
    if (!(strcmp(argv[1],"phi2"))){ f=phi2;
    if (!(strcmp(argv[2],"dphi1"))){ df=dphi1;
    if (!(strcmp(argv[2],"dphi2"))){ df=dphi2;
    val=atof(argv[3]);n=atoi(argv[4]);
```

```

double result=method(f,df,val,n,&p);
if (p) printf("x=%f\n",result);
else printf("The_process_diverges\n");
return 0;
}

```

Остается скомпилировать программу со всеми подключенными библиотеками. Для этой цели следует выполнить команду:

```
> gcc -o solver main.c -lmy -lm
```

После компиляции исполняемый файл будет иметь имя solver. На рис. 3.11 показаны примеры работы разработанного программного приложения с разными исходными параметрами.

```

juna@artamonov: ~/Документы/TeXSubject/Programming/Cursov/Code
Файл  Правка  Вид  Поиск  Терминал  Справка
juna@artamonov:~/Документы/TeXSubject/Programming/Cursov/Code$ gcc -o solver main.c -lmy -lm
juna@artamonov:~/Документы/TeXSubject/Programming/Cursov/Code$ ./solver phi1 dphi1 13 20
x= 12.003063
juna@artamonov:~/Документы/TeXSubject/Programming/Cursov/Code$ ./solver phi1 dphi1 3 20
The process diverges
juna@artamonov:~/Документы/TeXSubject/Programming/Cursov/Code$ ./solver phi2 dphi2 3 20
x= 1.337215
juna@artamonov:~/Документы/TeXSubject/Programming/Cursov/Code$ ./solver phi2 dphi2 13 20
The process diverges
juna@artamonov:~/Документы/TeXSubject/Programming/Cursov/Code$

```

Рис. 3.11. Примеры работы с исполняемым файлом

Для сборки статической библиотеки вместо компилятора gcc в ключом -shared необходимо просто заархивировать созданные объектные файлы в архив с расширением .a:

```
> ar r libmy.a func.o method.o
```

После этого выполнить общую компиляцию:

```
> gcc -o solver main.c -L. -lmy -lm
```

Здесь ключ -L. указывает, что библиотеку libm.a следует искать в текущей директории. После компиляции файл статической библиотеки libm.a можно удалить, так как она полностью добавляется в исполняемый файл, что делает его более автономным. Однако это приводит к увеличению его объема, не позволяет другим программам использовать эту библиотеку.

3.2.2 Другие методы приближенного нахождения корней уравнений

Существуют и другие приближенные методы нахождения корней уравнения $f(x) = 0$. Ниже дано описание некоторых из них.

Метод деления отрезка пополам (номер метода 1). Предположим, что на отрезке $[a, b]$ в точке $x_0 \in [a, b]$ график функции $f(x)$ пересекает ось абсцисс, то есть имеет место соотношение $f(x_0) = 0$ (см. рис. 3.12). Идея метода состоит в том, чтобы последовательно сдвигать левую или правую границу отрезка $[a, b]$ в точку $c = \frac{a+b}{2}$ в зависимости от знаков функции $f(x)$ на концах отрезка. Например, для представленного рис. 3.12: если $f(c) < 0$, то сдвигаем левую границу, то есть получаем новый отрезок $[c, b]$, если $f(c) > 0$, то сдвигаем правую границу, то есть получаем отрезок $[a, c]$. В итоге будет сохраняться условие $x_0 \in [c, b]$ или $x_0 \in [a, c]$. Процесс следует завершить, когда получится отрезок длиной меньше заданной ε . Следует учесть, что в реальном вычислительном процессе при малой величине ε может оказаться, что при вычислении по формуле $c = \frac{a+b}{2}$ точное значение c округлится до ближайшего a или b . В результате процедура вычисления корня зациклится.

Программа должна предусматривать возможность подобной ситуации, а также анализировать некорректные входные данные ($f(a)$ и $f(b)$ имеют один знак).

Метод касательных (номер метода 2). Идея метода продемонстрирована на рис. 3.13. Произвольно выбирается некоторая точка x_n , и проводится касательная к функции $f(x)$ в этой

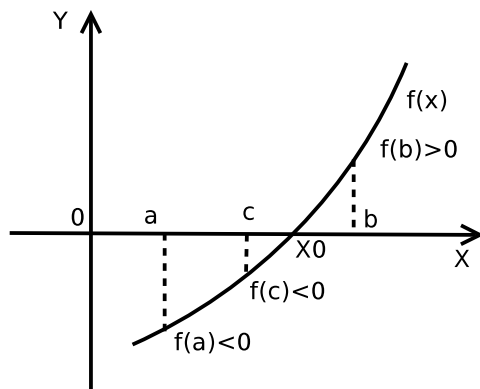


Рис. 3.12. Метод деления отрезка пополам

точке. Пусть касательная пересечет ось абсцисс в точке x_{n+1} , тогда проводится касательная к функции $f(x)$ уже в точке x_{n+1} . Данный процесс продолжают до достижения заданной точности.

Чтобы найти касательную, используется понятие производной. Известно, что производная в точке x_n равна тангенсу угла наклона касательной к функции $f(x)$ в этой точке.

Тогда из рис. 3.13 имеем:

$$f'(x_n) = \frac{f(x_n) - 0}{x_n - x_{n+1}} \Rightarrow x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

В качестве предварительного критерия окончания вычислений можно взять неравенство $|x_n - x_{n+1}| < \varepsilon$. Однако данное неравенство не гарантирует требуемой точности приближения x_{n+1} . Поэтому при выполнении указанного неравенства следует сделать дополнительную проверку знаков функции $f(x)$ на краях отрезка $[x_{n+1} - \varepsilon, x_{n+1} + \varepsilon]$. Если эта проверка даст одинаковые знаки на концах отрезка, то следует продолжить вычисление приближений, взяв меньшее ε .

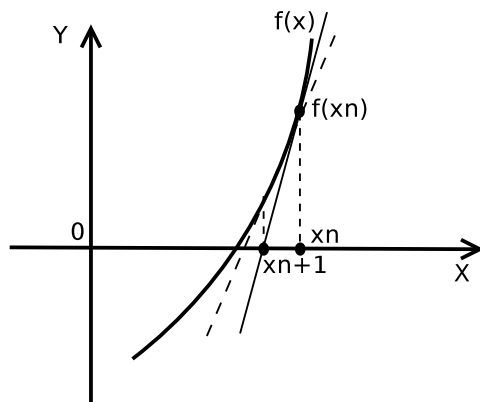


Рис. 3.13. Метод касательных

Если точность не достигается за значительное число итераций (например, 200), то вычисления целесообразно прервать и вернуть признак ошибки (например, -1).

Метод секущих (номер метода 3). Данный метод можно назвать модификацией от метода касательных (вместо производной берутся ее приближения в виде конечных разностей). Идея метода продемонстрирована на рис. 3.14.

Вначале выбираются два начальных приближения x_n и x_{n-1} , находятся значения функции $f(x)$ в этих точках. Затем через две точки с координатами $(x_{n-1}, f(x_{n-1}))$ и $(x_n, f(x_n))$ проводится прямая, которая пересечёт ось абсцисс в точке x_{n+1} . Тогда точка x_{n+1} становится следующим приближением $f(x_{n+1}) \approx 0$.

Из рис. 3.14 имеем:

$$\frac{f(x_n) - 0}{x_n - x_{n+1}} = \frac{0 - f(x_{n-1})}{x_{n+1} - x_{n-1}}.$$

Тогда после преобразований получаем:

$$x_{n+1} = \frac{f(x_n) \cdot x_{n-1} - f(x_{n-1}) \cdot x_n}{f(x_n) - f(x_{n-1})}.$$

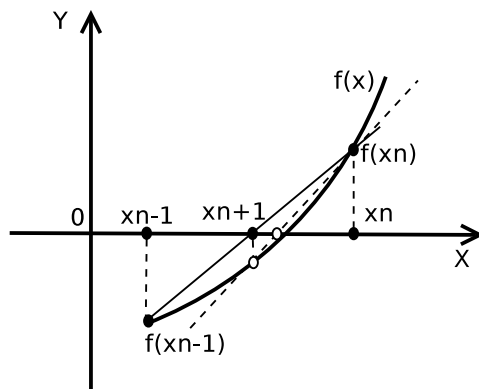


Рис. 3.14. Метод секущих

Обычно данную формулу преобразуют следующим образом:

$$x_{n+1} = \frac{x_n \cdot f(x_n) - x_n \cdot f(x_n) + f(x_n) \cdot x_{n-1} - f(x_{n-1}) \cdot x_n}{f(x_n) - f(x_{n-1})}.$$

Окончательно имеем:

$$x_{n+1} = x_n - f(x_n) \cdot \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}.$$

3.2.3 Задания для самостоятельной работы

1. Проведите тестирование методов деления пополам, касательных и секущих в соответствии с вариантом на примере решения уравнений:

1. $\sin(cx) - d = 0,$

2. $e^{cx} - d = 0,$

3. $\log_2(cx) - d = 0,$

$$4. \ x^3 + cx^2 + d = 0,$$

$$5. \ x^4 + cx^3 - dx = 0,$$

$$6. \ x^5 + cx^2 - d = 0$$

при разных значениях параметров c, d . Сравните число итераций этих методов при одном и том же значении точности. Придумайте три своих уравнения и используйте реализованные методы для нахождения их корней. Для оценки корректности получаемых решений целесообразно использовать рассмотренную систему символьных вычислений maxima или онлайн-калькулятор от Wolfram, доступный по ссылке <http://www.wolframalpha.com/>

Например, на рис. 3.15, 3.16 представлены графики, соответствующие решению уравнения $e^x - x^2 = 0$. В пояснительную записку требуется добавить скриншот таких графиков, а также найденное приближенное решение уравнения - на рис. 3.16 показаны точки пересечения графика с осью абсцисс.

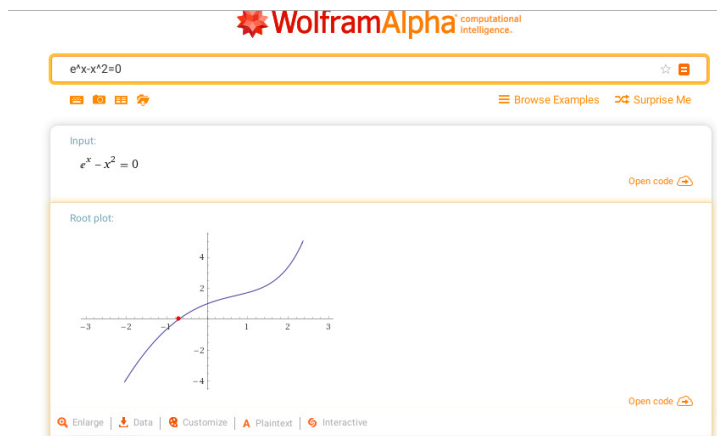


Рис. 3.15. Пример расчета в калькуляторе wolfram

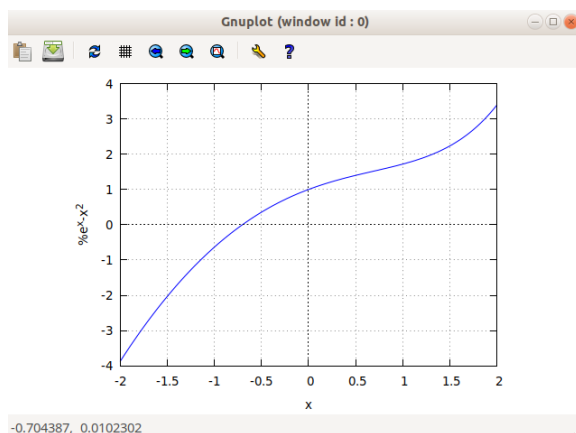


Рис. 3.16. Пример расчета в системе maxima

Распределение заданий по вариантам

Вариант:	1	2	3	4	5	6
Метод:	1,3	2,3	1,2	2,3	2,3	1,2
Уравнение:	1,4,5	1,5,6	2,4,5	1,2,3	1,2,6	3,4,6

Вариант:	7	8	9	10	11	12
Метод:	1,3	2,3	1,2	2,3	2,3	1,2
Уравнение:	2,3,4	1,2,3	3,4,5	4,5,6	1,5,6	1,2,6

Вариант:	13	14	15	16	17	18
Метод:	1,3	2,3	1,2	2,3	2,3	1,2
Уравнение:	1,4,6	2,5,6	3,5,6	1,3,5	2,4,6	1,3,5

Вариант:	19	20	21	22	23	24
Метод:	1,3	2,3	1,2	2,3	2,3	1,2
Уравнение:	1,5,6	2,3,5	1,4,5	2,3,6	2,4,5	2,4,5

4.1 Пример разработки игровой программы

При разработке игровой программы следует продемонстрировать навыки модульного проектирования программного приложения.

Для примера разработаем игровую программу «Змейка», суть которой состоит в том, что игрок управляет змеей, которая ползает по плоскости, ограниченной стенами, собирает предметы, от которых она становится или длинее, или короче в зависимости от их типа. Требуется избегать столкновения с собственным хвостом и краями игрового поля.

Декомпозируем задачу на следующие относительно независимые модули:

- Модуль 1. Формирование игрового поля, отображение игрового поля.
- Модуль 2. Формирование предметов, изменение состояния предметов, отображение предметов на игровом поле.
- Модуль 3. Построение структуры змеи, пересчет координат змеи при движении вверх, вниз, вправо, влево, расчет длины змеи при столкновении с предметом, визуализация змеи на игровом поле.
- Модуль 4. Основная программа, в которой комбинируется использование всех вышеперечисленных модулей.

При разработке игровой программы часто приходится обрабатывать события при нажатии на клавиши клавиатуры. Однако стандартная техника чтения с клавиатуры требует нажатия клавиши ввода (Enter), что в условиях игровой программы оказывается неудобно.

В таких условиях часто меняют стандартную технику взаимодействия с консолью. Одна из возможностей изменения режима взаимодействия с консолью - это использование библиотеки `ncurses`, которая обеспечивает кроссплатформенность, позволяет задавать координаты в виде знакомест и цвет выводимых символов. Подробное описание функционала данной библиотеки можно получить из официальной документации [4]. Здесь мы опишем лишь некоторые функции данной библиотеки, которые могут потребоваться в разрабатываемом программном приложении.

- `chtype` - тип символов, с которыми работает `ncurses` (он включает в себя код символа, цвет и дополнительные атрибуты).

Для работы с этим типом формат объявления следующий:

```
chtype c = 'a' | Special_code;
```

В качестве `Special_code` можно использовать `A_BLINK` (включает режим мерцания), `A_DIM` (пониженная яркость), `A_BOLD` (повышенная яркость), `A_NORMAL` (нормальное отображение), `A_UNDERLINE` (подчёркнутый),

`A_REVERSE` (инверсный).

- `initscr()` - функция, инициализирующая экран. С данной функции необходимо начинать использование библиотеки `ncurses`.
- `endwin()` - функция завершает работу с библиотекой `ncurses`, возвращая в стандартный режим взаимодействия с консолью.
- `move(y,x)` - курсор переносится в позицию с координатой (x,y) (обратите внимание, в `move` сначала нужно указать координату по оси y, а потом по оси x), при этом начало координат - это верхний левый угол, ось ординат направлена сверху вниз.
- `getmaxx(stdscr)`, `getmaxy(stdscr)` - функции возвращают максимальное значение координат x и y соответственно для данного экрана.
- `clear()` - очистка экрана.
- `noecho()`, `echo()` - соответственно функция `noecho` отключает автоматическое отображение при вводе. Функция `echo` отменяет действие функции `noecho`.
- `raw()` - запрещает обрабатывать все системные события клавиатуры, функция `noraw()` отключает этот режим.
- `cbreak()` - в данном режиме продолжают обрабатываться события `Ctrl+C`, `Ctrl+D`, `nocbreak` отменяет это действие.

- **keypad(stdscr, TRUE(FALSE))** - позволяет установить режим обработки функциональных клавиш, если вызвать функцию с параметром TRUE. Для отключения нужно передать FALSE во втором параметре. Вот зарезервированные названия командных клавиш стрелок: KEY_DOWN, KEY_UP, KEY_LEFT, KEY_RIGHT.
- **getyx(stdscr, int y, int x)** - получение текущих координат курсора (координаты записываются в переданные переменные x,y).
- **getch()** - функция возвращает введенный символ.
- **addch(chtype)** - выводит символ ch в текущую позицию курсора и перемещает курсор на один символ вправо или в начало следующей строки.
- **insch(chtype)** - вставка символа, все символы, стоящие после курсора, перемещаются на одну позицию вправо.
- **start_color()** - данная функция должна быть запущена перед работой с цветами символов.
- **init_pair(n, C_symbols, C_background)** - создание палитры, каждой палитре присваивается номер (n), цвет символов (C_symbols), цвет фона (C_background). В качестве возможных значений цветов можно использовать:
COLOR_BLACK, COLOR_RED, COLOR_GREEN,
COLOR_YELLOW, COLOR_BLUE, COLOR_MAGENTA,
COLOR_CYAN, COLOR_WHITE.
- **COLOR_PAIR(n)** - используется, чтобы указать номер палитры у символа. Например, ниже представлен фрагмент, демонстрирующий использование данной команды:

```
...
chtype c;
```

```

start_color();
init_pair(1, COLOR_CYAN, COLOR_WHITE);
c='w'|COLOR_PAIR(1);
...

```

- **printw()** - аналог функции `printf` в библиотеке `ncurses`.
- **scanw()** - аналог функции `scanf` в библиотеке `ncurses`.

В качестве примера ниже приведен код простейшей программы, которую можно рассматривать как «рисовалку». Программа перемещает курсор в соответствии с клавишами управления курсором, заливая текущее местоположения курсора двумя разными цветами: синим цветом, черным цветом (цветом фона). Цвета можно менять, можно очистить экран, выйти из программы. На рис. 4.1 представлен пример работы данной программы.

```

#include<ncurses.h>
int main(){
    initscr();
    start_color();
    init_pair(1, COLOR_BLUE, COLOR_BLUE);
    init_pair(2, COLOR_BLACK, COLOR_BLACK);
    int xmax=getmaxx(stdscr),
        ymax=getmaxy(stdscr);
    int x_current=xmax/2,y_current=ymax/2;
    raw();
    noecho();
    keypad(stdscr, TRUE);
    chtype space_color=' '|COLOR_PAIR(1),
        space_bground=' '|COLOR_PAIR(2),c;
    int p=1;
    move(0, 0);
    printw("q_ _quit\n");
    printw("c_ _clear_screen\n");
}

```

```

printw("z_-_change_color");
move(y_current, x_current);
while(c!='q'){
    c=getch();
    if (c==KEY_DOWN)
        y_current=(y_current+1)%ymax;
    if (c==KEY_UP) {
        y_current=(y_current-1)%ymax;
        if (y_current<0) y_current=ymax;}
    if (c==KEY_LEFT) {
        x_current=(x_current-1)%xmax;
        if (x_current<0) x_current=xmax;}
    if (c==KEY_RIGHT)
        x_current=(x_current+1)%xmax;
    if (c=='c') {
        clear();
        move(0, 0);
        printw("q_-_quit\n");
        printw("c_-_clear_screen\n");
        printw("z_-_change_color");
        x_current=xmax/2;
        y_current=ymax/2;
    }
    if (c=='z') p=(p+1)%2;
    move(y_current, x_current);
    if (p) addch(space_color);
    else addch(space_bground);
    refresh();
}
echo();
cbreak();
keypad(stdscr, FALSE);
endwin();
return 0;}

```

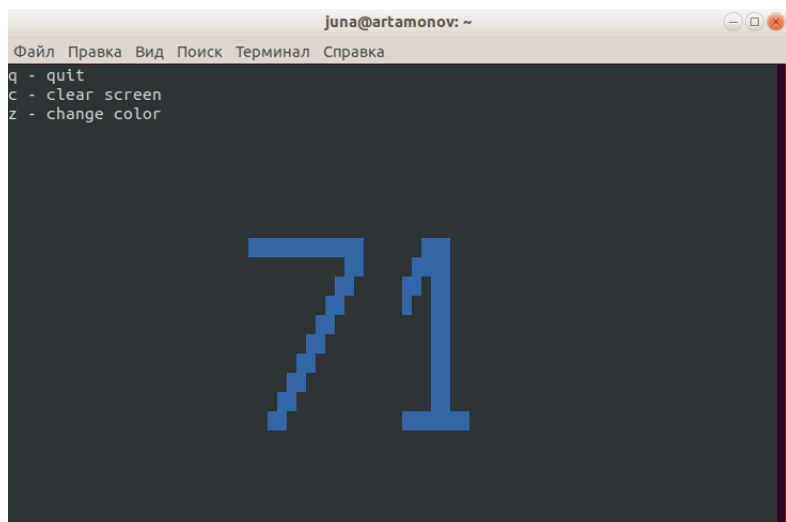



Рис. 4.1. Пример использования функционала библиотеки `ncurses`

Первое, о чем приходится задуматься при реализации игры «Змейка», – какие параметры описывают структуру змеи. Змея будет состоять из совокупности связанных знакомест, каждое из которых имеет определенные координаты по оси абсцисс и ординат. Также важно понимать, в какую сторону сейчас движется змея: вправо, влево, вверх, вниз. При расчете параметров движения будем считать, что змея каждый раз сдвигается на одно знакоместо в указанном направлении, поэтому требуется менять визуализацию только головы и хвоста (первого и последнего знакомест, обозначающих змею).

Препятствием следует считать любой предмет в игровом поле, координаты которого совпадают с координатами тела змеи, границу игрового поля, если голова змеи или какая-то часть ее тела попадают на эту границу, а также собственный хвост, ес-

ли координаты головы совпадают с координатами какого-либо знакоместа, обозначающего тело змеи.

Отдельное знакоместо удобно задавать структурой, хранящей координаты знакоместа, а также цветовую палитру знакоместа:

```
struct sign_place{
    int x;
    int y;
    int color;
};
```

Для хранения совокупности связанных знакомест можно использовать разные структуры: обычный статический массив, динамический массив, связанный список. Поскольку длина змеи, а значит, и количество знакомест меняется в ходе игры, использовать статический массив нерационально, его размер придется задавать с избытком, например, равным количеству знакомест всего игрового поля.

Наиболее подходящей структурой для реализации тела змеи является связанный список. В этом случае нет необходимости создавать отдельную структуру, хранящую координаты знакомест для змеи. Все координаты можно непосредственно хранить в узлах списка. Если все манипуляции по изменению длины змеи делать с головой списка, можно использовать односвязный список:

```
struct snake{
    int x;
    int y;
    int color;
    struct snake *next;
};
```

При реализации предмета будем учитывать следующие атрибуты: тип предмета (увеличивает или уменьшает длину змеи

при столкновении), время жизни предмета (если столкновения не произошло, по истечении времени жизни предмет исчезает), координаты и цвет знакоместа предмета, а также количество очков, на которое изменяется счет игры при съедении предмета. Все это удобно описать в виде структуры:

```
struct block{
    int x;
    int y;
    int color;
    int time;
    int type;
    char point;
};
```

Поскольку предметы появляются и исчезают во время игры, их удобно хранить в динамическом массиве из элементов описанной структуры block.

Наконец, игровое поле можно задать двумерным статическим массивом размерностью `getmaxx()` на `getmaxy()`, каждая ячейка которого хранит нуль или единицу. Единица обозначает границу игрового поля (препятствие).

Реализуем функцию, которая формирует игровое поле, автоматически обводя границы терминала, а также предусмотрим режим формирования игрового поля вручную, используя рассмотренную ранее программу «рисовалку». Данный функционал объединяется в модуле 1:

- Функция `automatic(...)` автоматически заполняет двумерный массив игрового поля `arr`, обводя границы по краям экрана.
- Функция `manual(...)` переводит программу в режим «рисовалки» игрового поля, запоминая изменения в массив `arr`.
- Функция `show(...)` отображает игровое поле по массиву `arr`.

Реализация описанных функций представлена ниже.

Листинг 4.1. field.c

```
void show_field(_Bool arr[][getmaxy(stdscr)],
               int xmax, int ymax){
    chtype border='_'|COLOR_PAIR(1);
    chtype body = '_'|COLOR_PAIR(2);
    for (int i=0;i<xmax;i++){
        for (int j=0;j<ymax;j++){
            if (arr[i][j])
            {
                move(j,i);
                addch(border);
            }
            else
            {
                move(j,i);
                addch(body);
            }
        }
    }
    refresh();
}

void automatic_field(_Bool arr[][getmaxy(stdscr)],
                    int xmax, int ymax){
    for (int i=0;i<ymax;i++)
    {
        for (int j=0;j<xmax;j++)
            if ((i==0)||(i==(ymax-1)))
                arr[j][i]=1;
        else
            if ((j==0)||(j==(xmax-1)))
                arr[j][i]=1;
    }
}
```

```

        else
            arr[j][i]=0;
    }
}

void manual_field(_Bool arr[][getmaxy(stdscr)],
                 int xmax, int ymax){
    for (int i=0;i<xmax;i++)
        for (int j=0;j<ymax;j++)
            arr[i][j]=0;
    int x_current=xmax/2,y_current=ymax/2;
    raw();
    noecho();
    keypad(stdscr, TRUE);
    chtype space_color='_'|COLOR_PAIR(1),
        space_bground='*'|COLOR_PAIR(2),c;
    int p=0;
    move(0, 0);
   printw("q_ _quit\n");
   printw("c_ _clear_screen\n");
   printw("z_ _change_color");
    move(y_current,x_current);
    while(c!='q'){
        c=getch();
        if (c==KEY_DOWN)
        {
            if (p==0) {move(y_current,x_current);
                addch('_'|COLOR_PAIR(9));refresh();}
            y_current=(y_current+1)%ymax;
        }
        if (c==KEY_UP) {
            if (p==0) {move(y_current,x_current);
                addch('_'|COLOR_PAIR(9));refresh();}
            y_current=(y_current-1)%ymax;
        }
    }
}

```

```

        if (y_current<0) y_current=ymax-1;}
if (c==KEY_LEFT) {
    if (p==0) {move(y_current,x_current);
        addch(' '|COLOR_PAIR(9));refresh();}
    x_current=(x_current-1)%xmax;
    if (x_current<0) x_current=xmax-1;}
if (c==KEY_RIGHT){
    if (p==0) {move(y_current,x_current);
        addch(' '|COLOR_PAIR(9));refresh();}
    x_current=(x_current+1)%xmax;}
if (c=='c') {
    clear();
    move(0, 0);
   printw("q_-_quit\n");
   printw("c_-_clear_screen\n");
   printw("z_-_change_color");
    x_current=xmax/2;
    y_current=ymax/2;
    for (int i=0;i<xmax;i++)
        for (int j=0;j<ymax;j++)
            arr[i][j]=0;
}
if (c=='z') p=(p+1)%2;
move(y_current, x_current);
if (p) {
    addch(space_color);
    arr[x_current][y_current]=1;
}
else {
    addch(space_bground);
    arr[x_current][y_current]=0;
}
refresh();
}
echo();

```

```

    noraw ();
    keypad(stdscr , FALSE);
}

```

При реализации функционала модуля 2 предусмотрим следующие действия:

- Функция `initial_blocks(...)` создает начальное множество предметов, случайным образом выбирая их координаты, тип, цвет и время существования.
- Функция `visible_blocks(...)` отображает созданные предметы на игровом поле.
- Функция `add_blocks(...)` добавляет к множеству уже существующих предметов новый предмет, инициализируя его координаты, тип, цвет и время существования случайным образом.
- Функция `delete_blocks(...)` удаляет из множества уже существующих предметов один последний предмет.
- Функция `work_blocks(...)` уменьшает время существования каждого предмета на единицу. Если время существования предмета равно нулю, он становится невидимым, в этом случае функция `work` случайно выбирает, включить его или нет. В случае включения для этого предмета опять случайно выбираются координаты, тип, цвет, время существования. Разные предметы могут находиться в одном месте игрового поля, приводя к множественным эффектам при столкновении со змеей.

Ниже представлена реализация описанных функций.

Листинг 4.2. `block.c`

```

void visible_blocks(struct block *arr , int count){
    ctype b;

```

```

    for (int i=0;i<count;i++){
        if (arr[i].time!=0){
            move(arr[i].y, arr[i].x);
            b=arr[i].point|COLOR_PAIR(arr[i].color);
            addch(b);
        }
    }
    refresh();
}

void initial_blocks(struct block **arr_blocks,
                    int *count, int xmax, int ymax,
                    int ttime, int color){
    int new_count=*count;
    if ((*arr_blocks)!=NULL)
    {
        while ((*count)>1)
            delete_blocks(arr_blocks, count);
        *arr_blocks=NULL;
    }
    (*count)=new_count;
    *arr_blocks=
        (struct block *)
        calloc(*count, sizeof(struct block));
    srand(time(NULL));
    for (int i=0;i<(*count);i++){
        (*arr_blocks)[i].x=1+rand()%(xmax-2);
        (*arr_blocks)[i].y=1+rand()%(ymax-2);
        (*arr_blocks)[i].color=1+rand()%(color);
        (*arr_blocks)[i].time=1+rand()%(ttime);
        (*arr_blocks)[i].type=rand()%(2);
        (*arr_blocks)[i].point=49+rand()%(9);
    }
}

void work_blocks(struct block **arr_blocks, int count,
                 int xmax, int ymax, int ttime, int color){

```



```

srand(time(NULL));
for (int i=0;i<count;i++){
    if ((*arr_blocks)[i].time>0)
        ((*arr_blocks)[i].time)--;
    else
        if (rand()%2){
            (*arr_blocks)[i].x=1+rand()%(xmax-2);
            (*arr_blocks)[i].y=1+rand()%(ymax-2);
            (*arr_blocks)[i].color=1+rand()%color;
            (*arr_blocks)[i].time=1+rand()%ttime;
            (*arr_blocks)[i].type=rand()%2;
            (*arr_blocks)[i].point=49+rand()%9;
        }
}
}

void add_blocks(struct block **arr_blocks, int *count,
               int xmax, int ymax, int ttime, int color){
    *arr_blocks=
        (struct block *)
        realloc(*arr_blocks,
                (++(*count))*sizeof(struct block));
    srand(time(NULL));
    (*arr_blocks)[*count-1].x=1+rand()%(xmax-2);
    (*arr_blocks)[*count-1].y=1+rand()%(ymax-2);
    (*arr_blocks)[*count-1].color=1+rand()%color;
    (*arr_blocks)[*count-1].time=1+rand()%ttime;
    (*arr_blocks)[*count-1].type=rand()%2;
    (*arr_blocks)[*count-1].point=49+rand()%9;
}

void delete_blocks(struct block **arr_blocks,
                  int *count){
    if (*count>1)
        *arr_blocks=
        (struct block *)realloc(*arr_blocks,
                                (--(*count))*sizeof(struct block));}

```

При реализации модуля 3 предусмотрим следующие функции:

- Функция `initial_snake(...)` создает одно звено змеи, если змея была создана до вызова этой функции, старая змея удаляется.
- Функция `add_snake(...)` увеличивает тело змеи на одно звено в голове.
- Функция `delete_snake(...)` удаляет последнее звено змеи в хвосте.
- Функция `move_snake(...)` перемещает тело змеи, начиная с головы с учетом направления движения вверх, вниз, вправо, влево. При этом происходит переприсваивание координат остальных звеньев с последовательным сдвигом их от головы до хвоста.
- Функция `show_snake(...)` отображает змею на игровом поле.
- Функция `conflict_body(...)` проверяет столкновение тела змеи с предметом, при столкновении изменяется длина змеи, счет игры, а также исключается сам предмет (точнее, время его существования устанавливается в нуль).
- Функция `conflict_field(...)` проверяет, произошло ли столкновение с препятствием.

Реализация данных функций представлена ниже.

Листинг 4.3. snake.c

```
void move_snake(struct snake *body, char direct ,
                int xmax,int ymax){
    struct snake *current=body;
    int x,y,x_old,y_old;
    x_old=current->x;y_old=current->y;
```

```

    if (direct=='R'){
        current->x=(body->x+1)%xmax;
        current->y=body->y;
    }
    if (direct=='L'){
        if ((body->x-1)>0) current->x=(body->x-1)%xmax;
        else current->x=xmax-1;
        current->y=body->y;
    }
    if (direct=='U'){
        if ((body->y-1)>0) current->y=(body->y-1)%ymax;
        else current->y=ymax-1;
        current->x=body->x;
    }
    if (direct=='D'){
        current->y=(body->y+1)%ymax;
        current->x=body->x;
    }
    while (current->next!=NULL){
        current=current->next;
        x=current->x;y=current->y;
        current->x=x_old;current->y=y_old;
        x_old=x;y_old=y;
    }
}

void initial_snake(struct snake **body,
                   int x, int y, int color){
    if ((*body)!=NULL){
        while ((*body)->next !=NULL) delete_snake(*body);
        free(*body);
    }
    *body=(struct snake *)malloc(sizeof(struct snake));
    (*body)->x=x;
    (*body)->y=y;
    (*body)->color=color;
    (*body)->next=NULL;
}

```

```

void delete_snake(struct snake *body){
    struct snake *current=body, *pre_current;
    if (current->next !=NULL){
        while(current->next != NULL) {
            pre_current=current;
            current=current->next;
        }
        pre_current->next=NULL;
        free(current);
    }
}

void add_snake(struct snake **body, char direct,
               int xmax, int ymax){
    struct snake *current=
        (struct snake *)malloc(sizeof(struct snake));
    current->color=(*body)->color;
    current->next=(*body);
    if (direct=='R'){
        current->x=((*body)->x+1)%xmax;
        current->y=(*body)->y;
    }
    if (direct=='L'){
        if (((*body)->x-1)>0)
            current->x=((*body)->x-1)%xmax;
        else current->x=xmax-1;
        current->y=(*body)->y;
    }
    if (direct=='U'){
        if (((*body)->y-1)>0)
            current->y=((*body)->y-1)%ymax;
        else current->y=ymax-1;
        current->x=(*body)->x;
    }
    if (direct=='D'){
        current->y=((*body)->y+1)%ymax;
        current->x=(*body)->x;
    }
}

```

```

    }
    *body=current;
}

void show_snake(struct snake *body){
    struct snake *current=body;
    chtype body_color;
    do{
        move(current->y, current->x);
        body_color='_'|COLOR_PAIR(current->color);
        addch(body_color);
        current=current->next;
    } while(current!=NULL);
}

_Bool conflict_body(struct snake *body){
    struct snake *current_one, *current_two;
    _Bool p=FALSE;
    current_one=body; current_two=body->next;
    while (current_one->next !=NULL){
        while (current_two != NULL){
            if (current_one->x==current_two->x &&
                current_one->y==current_two->y)
            {p=TRUE;
             return p;
            }
            current_two=current_two->next;
        }
        current_one=current_one->next;
        if (current_one != NULL)
            current_two=current_one->next;
    }
    return p;
}

void conflict_block(struct snake **body,
                   struct block *arr,

```

```

int count, char direct,
int xmax, int ymax,
int *count_points){
struct snake *current=*body;
int k=0;
while (current!=NULL){
    for (int i=0;i<count;i++){
        if (arr[i].x==current->x &&
            arr[i].y==current->y &&
            arr[i].time!=0) {
            if (arr[i].type)
                {k++;
                 *count_points=(*count_points)+
                               (arr[i].point-48);}
            else {k--;
                 *count_points=(*count_points)+
                               (arr[i].point-48);}
            arr[i].time=0;
        }
    }
    current=current->next;
}
if (k<0)
for(int i=0;i<(-1)*k;i++)
    delete_snake(*body);
else
for (int i=0;i<k;i++)
    add_snake(body,direct , xmax, ymax);
}

_Bool conflict_field(struct snake *body,
                    _Bool arr[][getmaxy(stdscr)],
                    int xmax, int ymax){
    _Bool p=FALSE;
    struct snake *current=body;
    while (current!=NULL){
        if (arr[current->x][current->y]==TRUE) {
            p=TRUE;

```

```

        return p;
    }
    current=current->next;
}
return p;
}

```

Для сопряжения первого, второго и третьего модулей вынесим их функции, а также используемые в них структуры в отдельные заголовочные файлы: `module_one.h`, `module_two.h`, `module_three.h`. Состав данных заголовочных файлов представлен ниже. Заметим, что функции какого-либо модуля могут использовать функции из других модулей. Например, в функции `conflict_block` модуля `snake.c` используется структура `block` из модуля `module_two.h`.

В основной программе `main.c` мы подключим все модули через заголовочные файлы `module_one.h`, `module_two.h`, `module_three.h`, в результате чего возможно многократное описание структуры `block` (сначала через заголовочный файл `module_two.h`, а потом через директивы препроцессора файла `snake.c`). Чтобы этого не происходило, используется специальная условная конструкция:

```

#ifndef h_block
#define h_block
...
#endif

```

Здесь проверяется условие, что в окружении не определена переменная `h_block`. Если это так, то переменная определяется, и код (отмечен выше многоточием) выполняется, в противном случае код игнорируется. Все это защищает от повторных переопределений функций и структур данных.

Листинг 4.4. `module_one.h`

```

void automatic_field(_Bool [][getmaxy(stdscr)],
                    int, int);

```

```

void manual_field(_Bool [][]getmaxy(stdscr)),
                 int, int);
void show_field(_Bool [][]getmaxy(stdscr)),
               int, int);

```

Листинг 4.5. module_two.h

```

#ifndef h_block
#define h_block

struct block{
    int x;
    int y;
    int color;
    int time;
    int type;
    char point;
};

#endif

void initial_blocks(struct block **,
                   int *, int, int, int, int);
void add_blocks(struct block **,
               int *, int, int, int, int);
void delete_blocks(struct block **, int *);
void work_blocks(struct block **,
                int, int, int, int, int);
void visible_blocks(struct block *, int );

```

Листинг 4.6. module_three.h

```

#ifndef h_snake
#define h_snake

struct snake{
    int x;
    int y;

```



```

    int color;
    struct snake *next;

};

#endif

void initial_snake(struct snake **, int, int, int);
void show_snake(struct snake *);
void add_snake(struct snake **, char, int, int);
void delete_snake(struct snake *);
void move_snake(struct snake *, char, int, int);
_Bool conflict_body(struct snake *);
void conflict_block(struct snake **,
    struct block *, int, char, int, int, int *);
_Bool conflict_field(struct snake *,
    _Bool [][][getmaxy(stdscr)], int, int);

```

После написания основных модулей к ним необходимо подключить требуемые заголовочные файлы. Ниже представлены листинги подключения заголовочных файлов для всех разработанных модулей.

Листинг 4.7. Подключение заголовочных файлов для field.c

```
#include <ncurses.h>
```

Листинг 4.8. Подключение заголовочных файлов для block.c

```
#include <stdlib.h>
#include <ncurses.h>
#include <time.h>
#include "module_two.h"
```

Листинг 4.9. Подключение заголовочных файлов для snake.c

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

```
#include<stdio.h>
#include<time.h>
#include<unistd.h>
#include "module_two.h"
#include "module_three.h"
```

Теперь можно перейти к компиляции всех модулей. Обратите внимание, в файлах field.c, block.c, snake.c отсутствует функция main, из этих файлов нужно создать объектные файлы без их линковки со всеми библиотеками, отложив этап линковки до компиляции основного файла main.c. Ниже представлены требуемые команды для компилятора:

```
>gcc -c field.c -lcurses
>gcc -c block.c
>gcc -c snake.c
```

В результате выполнения этих команд в рабочем каталоге будут созданы объектные файлы field.o, block.o, snake.o. После этого можно переходить к разработке четвертого модуля, в котором будет реализована главная функция main. Ниже представлен листинг данного модуля.

Листинг 4.10. Файл main.c

```
#include<stdio.h>
#include<stdlib.h>
#include<ncurses.h>
#include<time.h>
#include<unistd.h>
#include"module_one.h"
#include"module_two.h"
#include"module_three.h"

int main(){
    initscr();
    _Bool start_game=FALSE;
    int blocks_p;
    struct block *arr_block=NULL;
```

```

int xmax=getmaxx(stdscr),
    ymax=getmaxy(stdscr),
    count_block=5;
_Bool arr_field[xmax][ymax];
char direct='R';
int points=0, count_conflict=3;
int speed=70000, count_body=5;
int time_life=70;
struct snake *body=NULL;
start_color();
init_pair(1, COLOR_WHITE, COLOR_WHITE);
init_pair(2, COLOR_WHITE, COLOR_BLACK);
init_pair(3, COLOR_BLACK, COLOR_RED);
init_pair(4, COLOR_BLACK, COLOR_GREEN);
init_pair(5, COLOR_BLACK, COLOR_YELLOW);
init_pair(6, COLOR_BLACK, COLOR_BLUE);
init_pair(7, COLOR_BLACK, COLOR_MAGENTA);
init_pair(8, COLOR_BLACK, COLOR_CYAN);
init_pair(9, COLOR_BLACK, COLOR_BLACK);
raw();
noecho();
curs_set(0);
timeout(-1);
chtype c='_';
keypad(stdscr, TRUE);

while(c!='q'){
    timeout(-1);
    clear();
    move(1,2);
    addstr("q-Exit\n");
    move(2,2);
    addstr("1- automatic_field\n");
    move(3,2);
    addstr("2- manual_field\n");
    move(4,2);
    addstr("3- new_game\n");
    move(5,2);

```

```

addstr("4_ _continue_game\n");
switch (c){
case '1':
    automatic_field(arr_field , xmax,ymax);
    clear();
    show_field(arr_field ,xmax, ymax);
    break;
case '2':
    clear();
    manual_field(arr_field ,xmax,ymax);
    raw();
    noecho();
    keypad(stdscr , TRUE);
    timeout(0);
    break;
case '3':
    start_game=TRUE;
    count_block=5;
    initial_blocks(&arr_block , &count_block ,
                    xmax, ymax, time_life , 8);
    initial_snake(&body,xmax/2,ymax/2,5);
    points=0;
    for (int i=0;i<count_body;i++)
        add_snake(&body,direct , xmax,ymax);
    timeout(0);
    while (c!='q'){
        clear();
        show_field(arr_field , xmax,ymax);
        work_blocks(&arr_block , count_block ,
                    xmax, ymax, time_life , 8);
        visible_blocks(arr_block , count_block);
        move(0,0);
        printf("Points: %d; _q_ _exit ;
a_ _less_speed; _z_ _more_speed", points);
        if (c==KEY_DOWN) direct='D';
        if (c==KEY_UP) direct='U';
        if (c==KEY_LEFT) direct='L';
        if (c==KEY_RIGHT) direct='R';

```

```

if (c=='a') speed=(int)(speed*1.1);
if (c=='z') speed=(int)(speed*0.9);
if (c==KEY_RESIZE) {
    echo();
    noraw();
    keypad(stdscr, FALSE);
    endwin();
    printf("Game_over!\n");
    return 0;
}
move_snake(body,direct, xmax,ymax);
conflict_block(&body, arr_block, count_block,
               direct, xmax, ymax, &points);
if (conflict_field(body, arr_field, xmax, ymax)
|| conflict_body(body)) points--count_conflict;
show_snake(body);
blocks_p=rand()%10;
if (blocks_p>=6 && blocks_p<8)
    add_blocks(&arr_block, &count_block,
               xmax, ymax, time_life, 8);
else
    if (blocks_p>=8)
        delete_blocks(&arr_block, &count_block);
refresh();
usleep(speed);
c=getch();
}
break;

case '4':
    if (start_game){
        timeout(0);
        while (c!='q'){
            clear();
            show_field(arr_field, xmax,ymax);
            work_blocks(&arr_block, count_block,
                       xmax, ymax, time_life, 8);
            visible_blocks(arr_block, count_block);

```

```

        move(0,0);
       printw(" Points: %d", points);
a_less_speed; z_more_speed", points);
        if (c==KEY_DOWN) direct='D';
        if (c==KEY_UP) direct='U';
        if (c==KEY_LEFT) direct='L';
        if (c==KEY_RIGHT) direct='R';
        if (c=='a') speed=(int)(speed*1.1);
        if (c=='z') speed=(int)(speed*0.9);
        if (c==KEY_RESIZE) {
            echo();
            noraw();
            keypad(stdscr, FALSE);
            endwin();
            printf("Game_over!\n");
            return 0;
        }
        move_snake(body, direct, xmax,ymax);
        conflict_block(&body, arr_block,
            count_block, direct, xmax, ymax, &points);
if (conflict_field(body, arr_field, xmax, ymax)
|| conflict_body(body)) points-=count_conflict;
        show_snake(body);
        blocks_p=rand()%10;
        if (blocks_p>=6 && blocks_p<8)
            add_blocks(&arr_block, &count_block,
                xmax, ymax, time_life, 8);
        else
            if (blocks_p>=8)
                delete_blocks(&arr_block, &count_block);
        refresh();
        usleep(speed);
        c=getch();
    }
}
break;
default:
    break;

```

```

    }
    c=getch ();
    if (c==KEY_RESIZE) {
        echo ();
        noraw ();
        keypad(stdscr , FALSE);
        endwin ();
        printf("Game_over!\n");
        return 0;
    }
}
echo ();
noraw ();
keypad(stdscr , FALSE);
endwin ();
return 0;
}

```

Для компиляции всего проекта необходимо выполнить следующую команду:

```
>gcc main.c field.o block.o snake.o -lcurses
```

После выполнения данной команды в рабочей директории будет создан исполняемый файл a.out.

Когда программа состоит из множества различных модулей, часто приходится выполнять однообразную и объемную работу по их компиляции и сопряжению. В этом случае на помощь приходит утилита make, которая позволяет автоматизировать процесс сборки сложного проекта, состоящего из отдельных модулей. Например, в нашем случае сборка всего проекта состоит из этапа создания объектных файлов field.o, block.o, snake.o, а затем их совместной компиляции вместе с главным модулем main.c.

Утилита make работает с файлом Makefile. Данный файл должен находиться в каталоге, из которого доступны все сопрягаемые модули. Файл Makefile имеет специальный формат, состоящий из целей. Каждая цель пишется с новой строки и обозначается

ется словом, которое будет потом вызываться совместно с утилитой `make`. После имени цели ставится двоеточие, после которого перечисляются файлы, используемые для достижения цели, в качестве таких имен можно применять имена других целей, если для реализации искомой цели необходимо вначале получить результат этих других целей. Далее после названия цели, двоеточия и перечисления файлов и целей с новой строки и обязательного одного знака табуляции необходимо написать перечень команд, которые будут выполняться для достижения результата цели. Исходя из такого описания, Makefile нашего проекта может иметь вид, представленный ниже.

Листинг 4.11. Makefile всего проекта

```
field: field.c
    gcc -c field.c -o field.o

block: block.c
    gcc -c block.c -o block.o

snake: snake.c
    gcc -c snake.c -o snake.o

game: main.c field block snake
    gcc main.c field.o block.o snake.o -o game
    -lncurses

run: game
    ./game

clean:
    rm -f *.o
```

Как видно из Makefile, имеется шесть целей, имена которых: `field`, `block`, `snake`, `game`, `run`, `clean`. Таким образом, утилиту `make` можно запускать со следующими командами:

- `>make field`

В результате данной команды будет создан объектный файл `field.o`.

- `>make block`

В результате данной команды будет создан объектный файл `block.o`.

- `>make snake`

В результате данной команды будет создан объектный файл `snake.o`.

- `>make game`

В результате данной команды будет создан исполняемый файл `game`. Поскольку для реализации этой цели указаны все предыдущие цели `field`, `block`, `snake`, то, если объектные файлы отсутствуют, они будут созданы, т.е. соответствующая цель будет достигнута перед реализацией основной цели.

- `>make run`

В результате данной команды файл `game` будет исполнен, если файл отсутствует, то вначале он будет построен с выполнением всех зависящих от него подцелей.

- `>make clean`

В результате данной команды все созданные в данной директории объектные файлы будут удалены.

В целом утилита `make` - очень гибкое и удобное средство, получить ее более подробное описание можно из других источников [5, 8].

На рис. 4.2, 4.3, 4.4, 4.5 представлены основные экранные формы разработанной игровой программы.

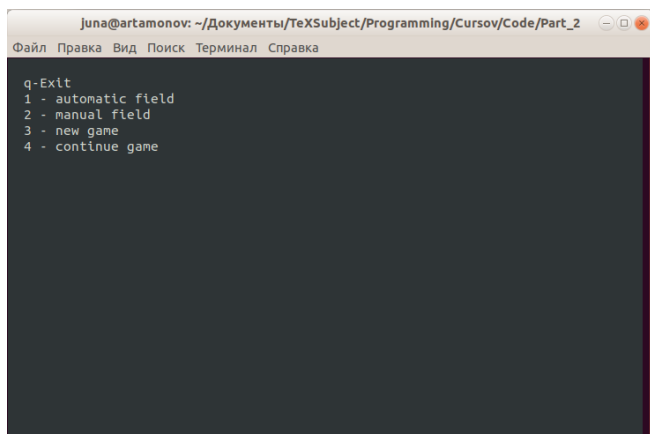


Рис. 4.2. Начальная экранная форма приложения

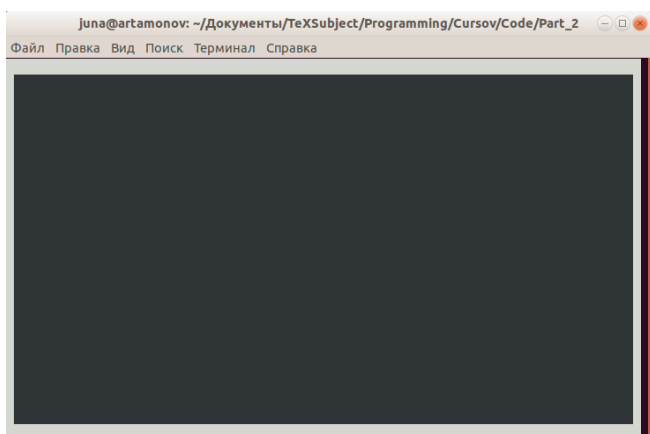


Рис. 4.3. Экранная форма при выборе первого пункта меню (автоматическое формирование границ экрана)



Рис. 4.4. Экранная форма при выборе второго пункта меню (ручное формирование границ экрана)



Рис. 4.5. Экранная форма при выборе третьего пункта меню (режим игры)

4.2 Система управления версиями программного обеспечения git

Сложные программы всегда находятся в стадии развития: исправляются ошибки, добавляется новый функционал, исключается устаревший. В ходе таких изменений очень часто приходится делать откаты назад или пытаться развивать программный продукт по какому-либо другому альтернативному направлению, то есть делать ветвления (*fork* - форки). Для возможности отслеживания всех таких изменений, а также отката программы на любой момент фиксации изменений разработаны специальные системы управления версионностью. Такие системы в течение всего процесса разработки программы позволяют фиксировать любое изменение, сопровождая его комментариями, храня дату и автора изменений, а при необходимости предоставляют возможность вернуться к состоянию программы (восстановить содержимое всех файлов и папок) на заданный момент времени.

Если над программой работает несколько человек, то может возникнуть еще одна сложность - разные люди могут пытаться вносить противоречивые изменения в одно и то же место программы, такие ситуации называются конфликтами. Системы управления версиями программ должны предлагать эффективные методы борьбы с конфликтами.

Важным понятием при использовании систем управления версиями программ является понятие **репозитория**. Репозиторий можно воспринимать как обычный каталог с файлами и другими каталогами. Однако в репозитории содержится еще и мета-информация, связанная с хранением хронологии всех изменений. Такую мета-информацию можно воспринимать как некоторую базу данных внутри репозитория (обычно это специальный скрытый каталог, например, в системе *git* этот каталог имеет имя *.git*).

Принято различать два подхода к построению систем управления версиями программ: централизованные системы, распре-

деленные системы. Централизованные системы предполагают использование одного центрального репозитория, и все участники должны перед началом работы получить из него свежие данные, а по окончании работы записать изменения в центральный репозиторий.

Распределенные системы не требуют изначального выбора главного, центрального репозитория. Фактически репозитории всех участников равноправны и независимы, но при необходимости могут быть синхронизированы друг с другом. В настоящее время все большее признание получают именно распределенные системы, ярким представителем которых является система `git`, разработанная изначально Линусом Торвальдсом в 2005 году для сопровождения процесса создания ядра операционной системы Linux [8].

Рассмотрим основной функционал системы `git`.

Начнем с установки системы `git`. Здесь следует обратиться к официальному сайту <https://git-scm.com/>. В операционных системах семейства GNU/Linux (дистрибутивы Debian, Ubuntu) для установки можно использовать команду:

```
>sudo apt-get install git
```

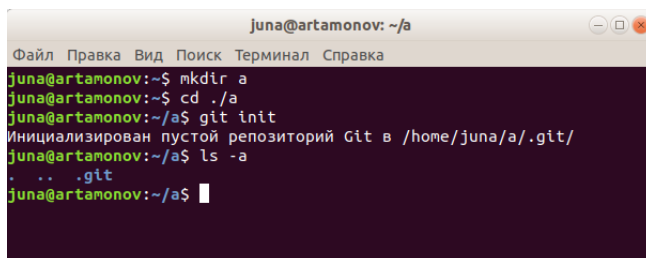
Работа с системой `git` начинается с создания репозитория. Для этого, перейдя в выбранный каталог, для которого необходимо создать репозиторий, следует выполнить команду:

```
>git init
```

После выполнения этой команды в текущей директории будет создан специальный скрытый каталог `.git`, который и будет обслуживать хронологию изменений текущего каталога. В качестве примера на рис. 4.6 создан пустой каталог с именем «а», внутри него инициализирован репозиторий `git`, а также показан состав каталога после инициализации, включая скрытые файлы. Для контроля состояния репозитория используется команда:

```
> git status
```

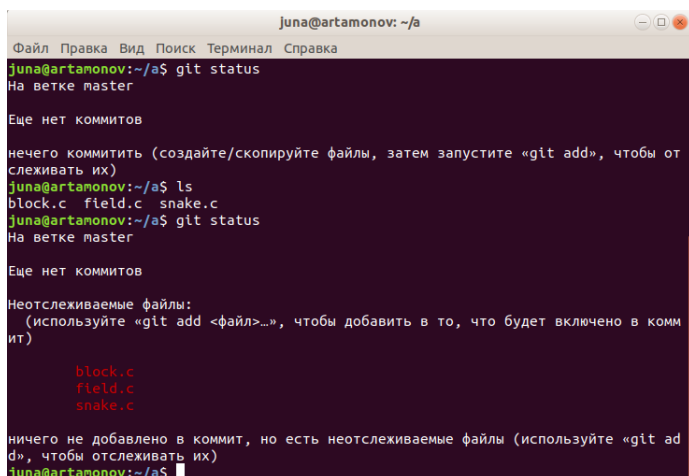
Если в каталоге нет никаких изменений, система `git` сообщит об этом или, наоборот, укажет, что относительно хранимой системой `git` истории, произошли изменения.



```
juna@artamonov: ~/a
Файл Правка Вид Поиск Терминал Справка
juna@artamonov:~$ mkdir a
juna@artamonov:~$ cd ./a
juna@artamonov:~/a$ git init
Инициализирован пустой репозиторий Git в /home/juna/a/.git/
juna@artamonov:~/a$ ls -a
.  ..  .git
juna@artamonov:~/a$
```

Рис. 4.6. Инициализация, создание репозитория

Добавим в текущий каталог файлы первого, второго и третьего модулей нашей игровой программы и повторно выполним команду `git status`. Результат выполнения показан на рис. 4.7 ниже.



```
juna@artamonov: ~/a
Файл Правка Вид Поиск Терминал Справка
juna@artamonov:~/a$ git status
На ветке master

Еще нет коммитов

нечего коммитить (создайте/скопируйте файлы, затем запустите «git add», чтобы отслеживать их)
juna@artamonov:~/a$ ls
block.c  field.c  snake.c
juna@artamonov:~/a$ git status
На ветке master

Еще нет коммитов

Неотслеживаемые файлы:
(используйте «git add <файл>...», чтобы добавить в то, что будет включено в коммит)

    block.c
    field.c
    snake.c

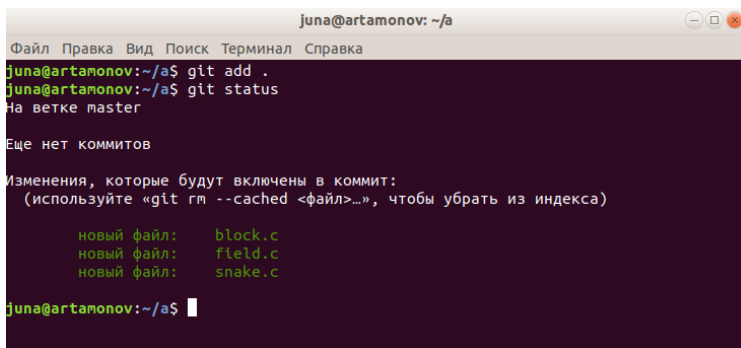
ничего не добавлено в коммит, но есть неотслеживаемые файлы (используйте «git add», чтобы отслеживать их)
juna@artamonov:~/a$
```

Рис. 4.7. Результат выполнения команды `git status` при добавлении файлов

Как видно из рисунка, система обнаружила новые файлы и в связи с этим рекомендует пользователю ряд действий.

Пока добавленные файлы система никак не отслеживает, мы не дали для этого ей никаких распоряжений. Чтобы добавить файлы к отслеживанию, нужно выполнить команду `git add` с перечислением имен всех файлов, которые необходимо отслеживать. Если требуется добавить все новые файлы, можно выполнить команду `git add .`

Ниже на рис. 4.8 показан результат выполнения данной команды.



```
juna@artamonov: ~/a
Файл Правка Вид Поиск Терминал Справка
juna@artamonov:~/a$ git add .
juna@artamonov:~/a$ git status
На ветке master

Еще нет коммитов

Изменения, которые будут включены в коммит:
(используйте «git rm --cached <файл>...», чтобы убрать из индекса)

    новый файл:   block.c
    новый файл:   field.c
    новый файл:   snake.c

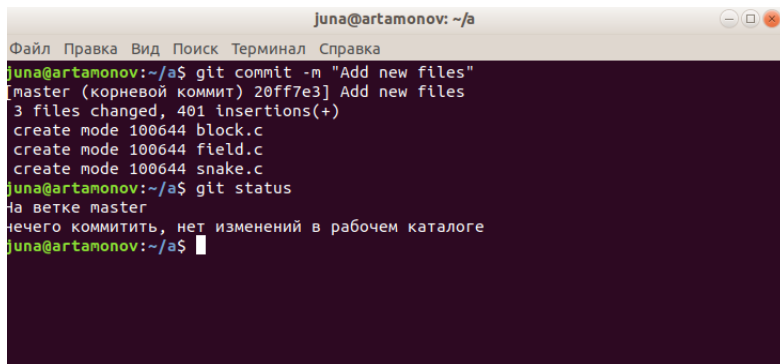
juna@artamonov:~/a$
```

Рис. 4.8. Результат выполнения команды `git add .`

Теперь система приняла файлы к учету, однако просит создать коммит.

Коммит - это важный термин системы `git`. По сути это обычный комментарий, однако у системы `git` к ним особое отношение: они являются обязательным атрибутом окончательной фиксации изменений в системе `git`. Коммит можно понимать как подтверждающее транзакционное действие. После внесения коммита откатить изменения будет невозможно.

Да, конечно, можно опять удалить добавленные файлы и закоммитить это изменение, но в хронологии системы эти файлы все равно останутся, и при необходимости их обратно можно получить, перейдя на нужный коммит. Для внесения коммита требуется выполнить команду `git commit -m name`. Ниже на рис. 4.9 представлен пример работы с данной командой.

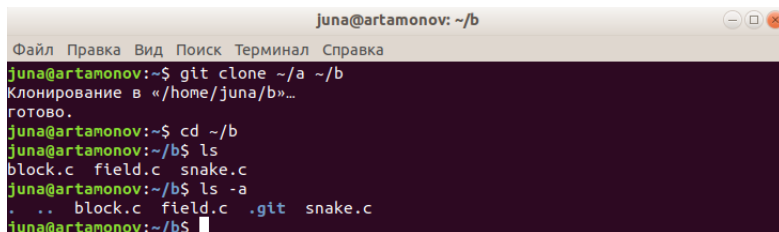


```
juna@artamonov: ~/a
Файл Правка Вид Поиск Терминал Справка
juna@artamonov:~/a$ git commit -m "Add new files"
[master (корневой коммит) 20ff7e3] Add new files
3 files changed, 401 insertions(+)
create mode 100644 block.c
create mode 100644 field.c
create mode 100644 snake.c
juna@artamonov:~/a$ git status
На ветке master
ничего коммитить, нет изменений в рабочем каталоге
juna@artamonov:~/a$
```

Рис. 4.9. Результат выполнения команды `git commit`

Еще одним важным действием в системе `git` является операция клонирования `git clone`. Чтобы выполнить клонирование локальной директории `/a` в локальную директорию `/b`, достаточно выполнить команду `git clone /a /b`. Ниже на рис. 4.10 представлен результат выполнения данной команды. Как видно из рисунка, содержимое каталога `/a` полностью скопировано в каталог `/b`, кроме того, там также появился скрытый каталог `.git`.

С этого момента каталоги `/a` и `/b` начинают самостоятельную жизнь - в них могут добавляться новые файлы или удаляться уже существующие и т.д., но при необходимости эти каталоги можно синхронизировать. Чтобы синхронизировать два репозитория, достаточно выполнить команду `git pull`.

A screenshot of a terminal window titled 'juna@artamonov: ~/b'. The window has a menu bar with 'Файл', 'Правка', 'Вид', 'Поиск', 'Терминал', and 'Справка'. The terminal shows the following commands and output:

```
juna@artamonov:~$ git clone ~/a ~/b
Клонирование в «/home/juna/b»...
готово.
juna@artamonov:~$ cd ~/b
juna@artamonov:~/b$ ls
block.c  field.c  snake.c
juna@artamonov:~/b$ ls -la
.  ..  block.c  field.c  .git  snake.c
juna@artamonov:~/b$
```

Рис. 4.10. Результат клонирования локальной директории

Однако сейчас каталоги `/a` и `/b` не являются совсем равноправными. Поскольку каталог `/b` был клонирован от каталога `/a`, он знает о его существовании (там есть специальное указание на так называемый remote репозиторий), в каталоге `/a` такой информации нет. Для примера добавим в каталог `/b` файл четвертого модуля `main.c`, а также файлы `module_one.h`, `module_two.h`, `module_three.h`, при этом только файл `main.c` добавим в систему `git` для отслеживания. В каталог `/a` добавим файл `Makefile`. После этого в каталоге `/b` выполним команду `git pull`. Поскольку в каталоге `/b` теперь отслеживается файл `main.c`, которого нет в каталоге `/a`, необходимо выполнить слияние (merge) данных. Нужно заметить, что слияние возможно, если оно непротиворечиво (например, в одном репозитории файл удалили, а в другом его же просто изменили). Для слияния автоматически возникает окно коммита, как показано на рис. 4.11 ниже.

После слияния мы увидим изменения, которые выполнены в репозитории `/a`. Данные изменения отражены на рис. 4.12 ниже.

Как видим, в каталоге `/b` появился файл `Makefile` из каталога `/a`.

Чтобы загрузить данные из каталога `/b` в каталог `/a`, нужно выполнить команду `git push`. Однако `git` по умолчанию не разрешает обновлять удаленный репозиторий, который создан посредством `git init`.

```
juna@artamonov: ~/b
Файл Правка Вид Поиск Терминал Справка
GNU nano 2.9.3 /home/juna/b/.git/MERGE_MSG

Merge branch 'master' of /home/juna/a

# Пожалуйста, введите сообщение коммита, для объяснения, зачем нужно
# это слияние, особенно, если это слияние обновленной вышестоящей
# ветки в тематическую ветку.
#
# Строки, начинающиеся с «#» будут проигнорированы, а пустое
# сообщение отменяет процесс коммита.

[ Read 8 lines ]
^G Помощь ^O Записать ^M Поиск ^K Вырезать ^J Выворнуть ^C ТекПозиц
^X Выход ^R ЧитФайл ^_ Замена ^U Отмен. выр ^T Словарь ^_ К строке
```

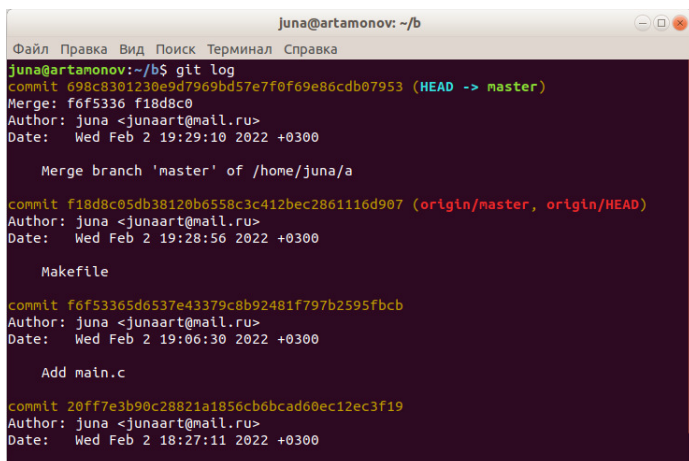
Рис. 4.11. Слияние данных

```
juna@artamonov:~/b$ git pull
remote: Подсчет объектов: 3, готово.
remote: Сжатие объектов: 100% (3/3), готово.
remote: Total 3 (delta 0), reused 0 (delta 0)
Распаковка объектов: 100% (3/3), готово.
Из /home/juna/a
 20ff7e3..f18d8c0 master -> origin/master
Merge made by the 'recursive' strategy.
 Makefile | 17 ++++++
 1 file changed, 17 insertions(+)
 create mode 100644 Makefile
juna@artamonov:~/b$ ls
block.c main.c module_one.h module_two.h
field.c Makefile module_three.h snake.c
juna@artamonov:~/b$
```

Рис. 4.12. Результат выполнения команды git pull

Репозиторий можно откатить до любого состояния, зафиксированного коммитом. Чтобы посмотреть все коммиты, можно использовать команду `git log`. На рис. 4.13 показан результат выполнения данной команды.

Чтобы откатить состояние каталога до какого-нибудь коммита, нужно использовать команду `git checkout` хеш коммита. Ниже на рис. 4.14 сначала показано текущее состояние директории `/b`, а далее состояние после отката по коммиту `20ff7e3b90c28821a1856cb6cad60ec12ec3f19`. Как видно из рисунка, добавленный файл `Makefile` уже отсутствует в каталоге `/b`.



```
juna@artamonov: ~/b
Файл  Правка  Вид  Поиск  Терминал  Справка

juna@artamonov:~/b$ git log
commit 698c8301230e9d7969bd57e7f0f69e86cdb07953 (HEAD -> master)
Merge: f6f5336 f18d8c0
Author: juna <junaart@mail.ru>
Date:   Wed Feb 2 19:29:10 2022 +0300

    Merge branch 'master' of /home/juna/a

commit f18d8c05db38120b6558c3c412bec2861116d907 (origin/master, origin/HEAD)
Author: juna <junaart@mail.ru>
Date:   Wed Feb 2 19:28:56 2022 +0300

    Makefile

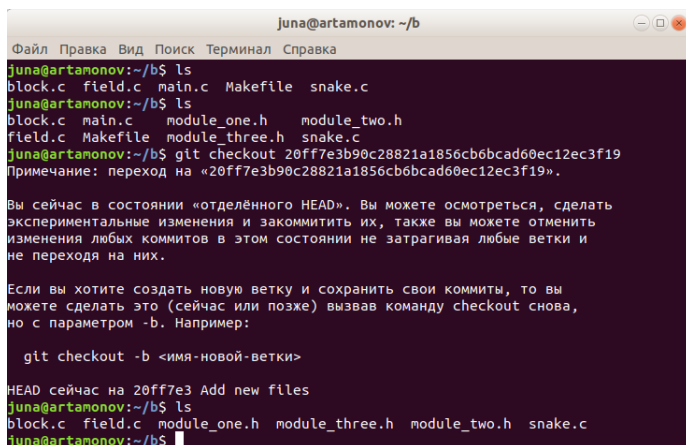
commit f6f53365d6537e43379c8b92481f797b2595fbc
Author: juna <junaart@mail.ru>
Date:   Wed Feb 2 19:06:30 2022 +0300

    Add main.c

commit 20ff7e3b90c28821a1856cb6cad60ec12ec3f19
Author: juna <junaart@mail.ru>
Date:   Wed Feb 2 18:27:11 2022 +0300
```

Рис. 4.13. Результат просмотра коммитов

Дальнейшие действия зависят от целей отката. Если целью было просто посмотреть содержимое каталога, то можно вернуться на последний коммит все той же командой `git checkout` хеш коммита или можно просто указать главную ветку `git checkout master`. Другая альтернатива может заключаться в том, что потребовалось сделать новую ветвь, в которой содержимое каталога будет следовать своей хронологии.



```
juna@artamonov: ~/b
Файл  Правка  Вид  Поиск  Терминал  Справка
juna@artamonov:~/b$ ls
block.c  field.c  main.c  Makefile  snake.c
juna@artamonov:~/b$ ls
block.c  main.c  module_one.h  module_two.h
field.c  Makefile  module_three.h  snake.c
juna@artamonov:~/b$ git checkout 20ff7e3b90c28821a1856cb6cad60ec12ec3f19
Примечание: переход на «20ff7e3b90c28821a1856cb6cad60ec12ec3f19».

Вы сейчас в состоянии «отделённого HEAD». Вы можете осмотреться, сделать
экспериментальные изменения и закоммитить их, также вы можете отменить
изменения любых коммитов в этом состоянии не затрагивая любые ветки и
не переходя на них.

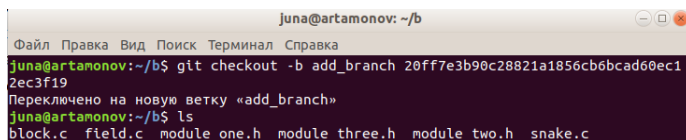
Если вы хотите создать новую ветку и сохранить свои коммиты, то вы
можете сделать это (сейчас или позже) вызвав команду checkout снова,
но с параметром -b. Например:

    git checkout -b <имя-новой-ветки>

HEAD сейчас на 20ff7e3 Add new files
juna@artamonov:~/b$ ls
block.c  field.c  module_one.h  module_three.h  module_two.h  snake.c
juna@artamonov:~/b$
```

Рис. 4.14. Результат отката до определенного коммита

Это можно сделать командой `git checkout -b имя-новой-ветки` хеш-коммита текущей ветки. Ниже на рис. 4.15 показана ситуация создания новой ветки с именем `add_branch` от коммита `20ff7e 3b90c28821a18 56cb6bcad60ec1 2ec3f19` ветки `master`. Как видно из рисунка, мы опять вернулись к состоянию репозитория на момент коммита `20ff7e3b90c28821a18 56cb6bcad60ec1 2ec3f19` (отсутствует файл `Makefile`), но теперь внутри этой ветки может быть своя параллельная хронология одного и того же каталога `/b`.



```
juna@artamonov: ~/b
Файл  Правка  Вид  Поиск  Терминал  Справка
juna@artamonov:~/b$ git checkout -b add_branch 20ff7e3b90c28821a1856cb6cad60ec1
2ec3f19
Переключено на новую ветку «add_branch»
juna@artamonov:~/b$ ls
block.c  field.c  module_one.h  module_three.h  module_two.h  snake.c
```

Рис. 4.15. Создание новой ветки

Для примера добавим в каталог /b объектные файлы field.o, block.o, snake.o, а также исполняемый файл game и посмотрим на содержимое каталога. Как видно из рис. 4.16, для ветки master мы имеем одно содержимое каталога /b, а для ветки add_branch другое содержимое. На рисунке также показано, что посмотреть список веток можно командой git branch -l.

```
juna@artamonov:~/b$ git checkout add_branch
Переключено на ветку «add_branch»
juna@artamonov:~/b$ ls
block.c  field.c  game      module_three.h  snake.c
block.o  field.o  module_one.h  module_two.h    snake.o
juna@artamonov:~/b$ git checkout master
Переключено на ветку «master»
Ваша ветка опережает «origin/master» на 2 коммита.
(используйте «git push», чтобы опубликовать ваши локальные коммиты)
juna@artamonov:~/b$ ls
block.c  field.c  main.c  Makefile  snake.c
juna@artamonov:~/b$ git branch --list
* add_branch
* master
juna@artamonov:~/b$
```

Рис. 4.16. Переключение между ветками

Использование всех возможностей системы git реализовано в различных хостингах IT-проектов. Одним из таких наиболее популярных хостингов является github.com. Для разработчика ПО его содержательный, востребованный контент на этом хостинге не только создает рекламу, заинтересованное окружение и возможность работать в команде, но и может способствовать его трудоустройству в передовую IT-компанию.

Рассмотрим использование github.com для хостинга IT-проектов. Для начала работы с github.com, очевидно, требуется пройти этап регистрации и создания собственного профиля. После этого в авторизованном режиме пользователю среди прочих действий предоставляется возможность создать свой приватный или публичный репозиторий.

Для примера на рис. 4.17 показан этап создания репозитория для хранения файлов разработанной игровой программы «Змейка». После нажатия на кнопку «Create repository» репозиторий будет создан.

The screenshot shows the GitHub 'Create repository' form. At the top, the 'Owner' is 'junaart' and the 'Repository name' is 'cursov_part_two', which is marked with a green checkmark. Below this is a tip: 'Great repository names are short and memorable. Need inspiration? How about [psychic-dollop?](#)'. The 'Description (optional)' field contains the text: 'Репозиторий содержит все файлы игровой программы "Змейка" в качестве демонстрационного примера'. Under the 'Visibility' section, the 'Public' radio button is selected, with the text 'Anyone on the internet can see this repository. You choose who can commit.' The 'Private' option is unselected. The 'Initialize this repository with:' section has three checked options: 'Add a README file' (with a sub-note: 'This is where you can write a long description for your project. [Learn more.](#)'), 'Add .gitignore' (with a sub-note: 'Choose which files not to track from a list of templates. [Learn more.](#)' and a dropdown menu showing 'gitignore template: C'), and 'Choose a license' (with a sub-note: 'A license tells others what they can and can't do with your code. [Learn more.](#)' and a dropdown menu showing 'License: GNU General Public ...'). At the bottom, it says 'This will set `main` as the default branch. Change the default name in your [settings](#).' A green 'Create repository' button is at the bottom.

Рис. 4.17. Создание репозитория на github.com

Начальный внешний вид созданного репозитория показан на рис. 4.18. Как видно из рисунка, репозиторий содержит три файла: файл `.gitignore` (включает шаблоны всевозможных временных файлов, которые порождаются компилятором языка C и которые не нужно портировать в репозиторий на github из локальных репозиториев), файл `LICENSE` (включает лицензию, которой отвечает содержимое репозитория, в нашем случае была выбрана лицензия GNU General Public License v3.0), файл `README.md` (описание назначения файлов репозитория). Как видно из рисунка, наш репозиторий имеет имя `cursov_part_two`. По факту пока это пустой репозиторий, не считая указанных файлов, которые были созданы автоматически из позиций, выбранных на рис. 4.17 при создании самого репозитория. Для локальной работы с репозиторием его необходимо клонировать.

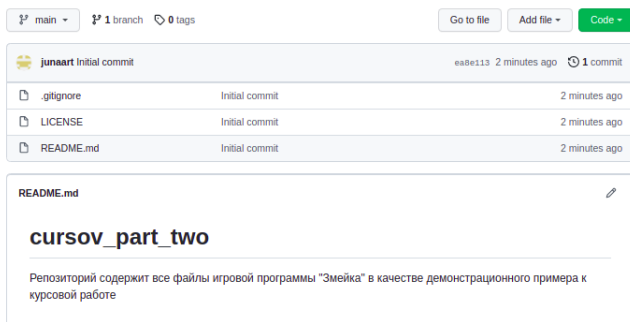


Рис. 4.18. Начальный внешний вид созданного репозитория

Клонировать репозиторий с github можно по нескольким протоколам, наиболее популярные из которых: https, ssh. Для работы по протоколу https необходимо сгенерить и получить токен доступа в разделе настроек своего профиля на сайте github. Для работы по протоколу ssh нужно на локальном компьютере установить систему ssh, далее по выбранному ассиметричному криптографическому протоколу (rsa, dsa и др.) сгенерировать публичный и секретный ключи, публичный ключ загрузить на сайт github, а для секретного ключа настроить его использование системой ssh. Здесь мы рассмотрим только использование протокола https. Для получения токена необходимо перейти в раздел [https:// github.com/settings/tokens](https://github.com/settings/tokens), внешний вид этого раздела показан на рис. 4.19. При нажатии на кнопку «Generate new token» после этапа повторной авторизации будет получено окно настроек доступа по токenu, внешний вид которого показан на рис. 4.20.

Как видно из рис. 4.20, здесь можно настроить срок использования токена, ввести описание токена, а также выбрать, какие функции доступны по токену (если мы хотим предоставить полный доступ по токену, необходимо выбрать галочками все позиции).

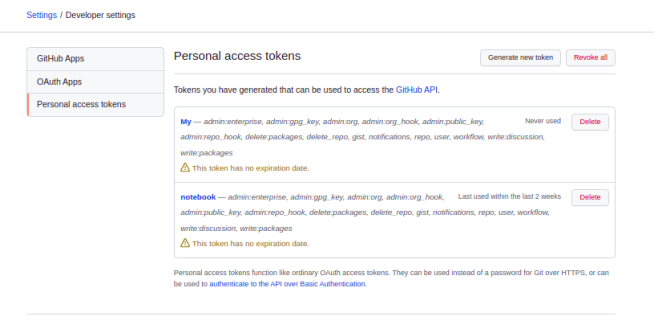


Рис. 4.19. Режим получения токена

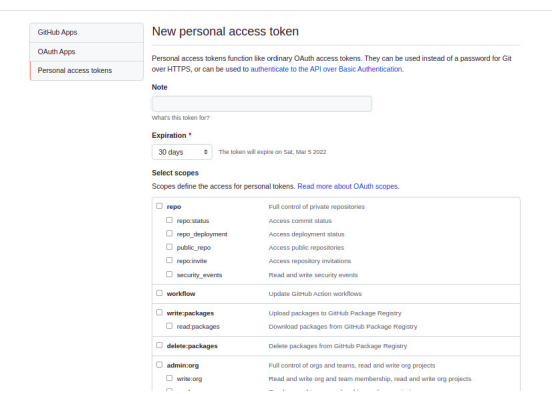


Рис. 4.20. Режим настройки доступа по токену

После нажатия на кнопку «Generate token» токен доступа будет сгенерирован, как показано на рис. 4.21.

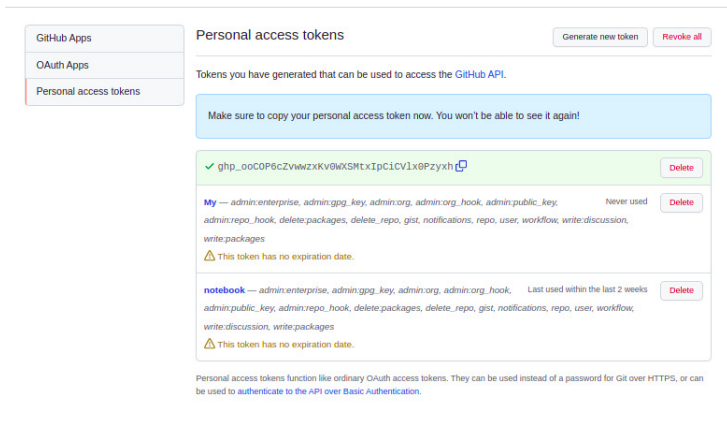


Рис. 4.21. Полученный токен

Данный токен нужно сохранить, например, в простом текстовом формате. После получения токена можно перейти к клонированию репозитория по протоколу https (заметим, что публичные репозитории можно клонировать и без токена, однако пушить изменения обратно на github без токена не получится). Для клонирования репозитория необходимо получить ссылку на репозиторий по соответствующему протоколу, как показано на рис. 4.22.

После получения ссылки в терминале своего локального компьютера выполняем команду:

```
> git clone https://github.com/junaart/cursov_part_two.git
```

Результат выполнения этой команды показан на рис. 4.23. После выполнения данной команды на локальном компьютере появится папка cursov_part_two. Добавим в нее все файлы игровой программы «Змейка» и запустим изменения на github.

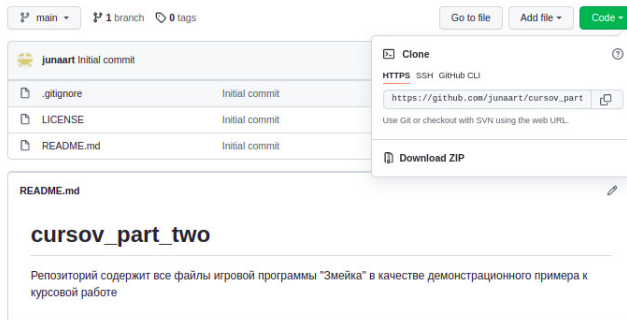


Рис. 4.22. Получение ссылки на репозиторий по протоколу https

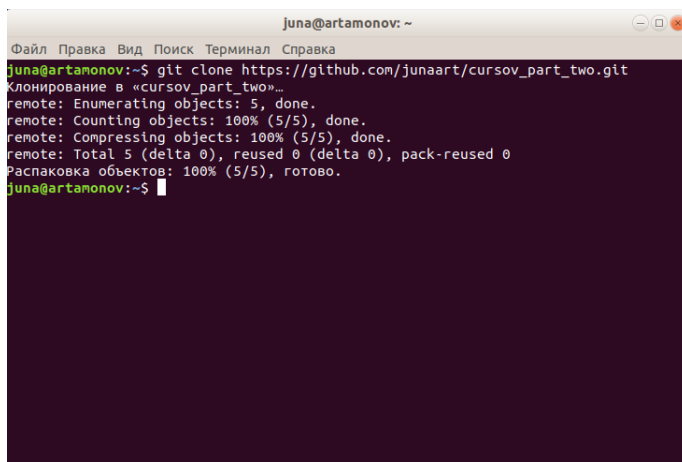
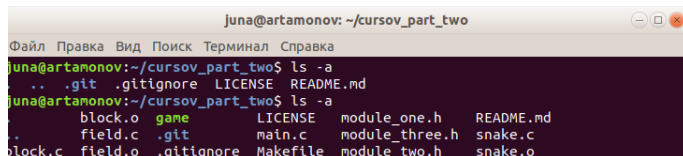


Рис. 4.23. Клонирование репозитория cursov_part_two на локальный компьютер

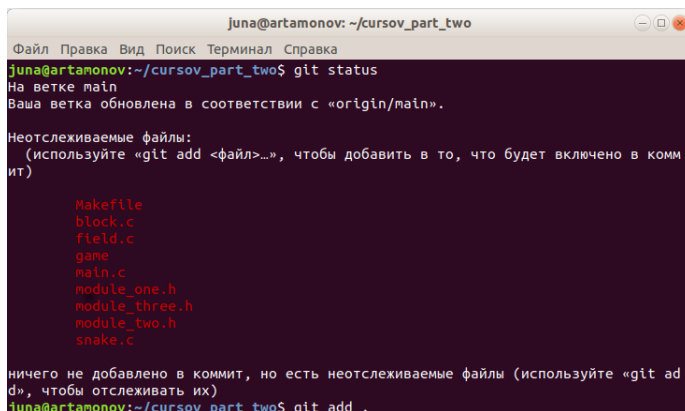
Данные действия показаны на рисунках ниже. На рис. 4.24 приведен состав директории непосредственно после ее клонирования с github, а также после добавления в нее файлов разработанной игровой программы.



```
juna@artamonov: ~/cursov_part_two
Файл Правка Вид Поиск Терминал Справка
juna@artamonov:~/cursov_part_two$ ls -a
.  ..  .git  .gitignore  LICENSE  README.md
juna@artamonov:~/cursov_part_two$ ls -a
.  ..  block.o  game  LICENSE  module_one.h  README.md
.  ..  field.c  .git  main.c  module_three.h  snake.c
.  ..  block.c  field.o  .gitignore  Makefile  module_two.h  snake.o
```

Рис. 4.24. Состав клонированной директории

На рис. 4.25 показан результат выполнения команд `git status` и `git add .`



```
juna@artamonov: ~/cursov_part_two
Файл Правка Вид Поиск Терминал Справка
juna@artamonov:~/cursov_part_two$ git status
На ветке main
Ваша ветка обновлена в соответствии с «origin/main».

Неотслеживаемые файлы:
(используйте «git add <файл>...», чтобы добавить в то, что будет включено в коммит)

    Makefile
    block.c
    field.c
    game
    main.c
    module_one.h
    module_three.h
    module_two.h
    snake.c

ничего не добавлено в коммит, но есть неотслеживаемые файлы (используйте «git add», чтобы отслеживать их)
juna@artamonov:~/cursov_part_two$ git add .
```

Рис. 4.25. Изменение состава клонированного репозитория

На рис. 4.26 представлен результат выполнения команд `git commit`, `git push`. При выполнении команды `git push` произойдет запрос логина и пароля для доступа к репозиторию на github.

Здесь необходимо ввести имя своего профиля при создании и скопированный токен соответственно.

```
juna@artamonov:~/cursov_part_two$ git commit -m "Добавление всех файлов проекта"
[main dd64f08] Добавление всех файлов проекта
9 files changed, 637 insertions(+)
create mode 100644 Makefile
create mode 100644 block.c
create mode 100644 field.c
create mode 100755 game
create mode 100644 main.c
create mode 100644 module_one.h
create mode 100644 module_three.h
create mode 100644 module_two.h
create mode 100644 snake.c
juna@artamonov:~/cursov_part_two$ git push
Username for 'https://github.com': junaart
Password for 'https://junaart@github.com':
Подсчет объектов: 11, готово.
Delta compression using up to 4 threads.
Сжатие объектов: 100% (11/11), готово.
Запись объектов: 100% (11/11), 12.55 KiB | 2.09 MiB/s, готово.
Total 11 (delta 0), reused 0 (delta 0)
To https://github.com:junaart/cursov_part_two.git
   ea8e113..dd64f08  main -> main
juna@artamonov:~/cursov_part_two$
```

Рис. 4.26. Сохранение локальных изменений на github

После выполнения этих действий содержимое репозитория на github обновится до состояния локальной репозитории, как показано на рис. 4.27. С этими репозиториями можно продолжить работать автономно. Допустим, другой пользователь скопировал текущее состояние репозитория на github, удалил у себя локально исполняемый файл `game` и сохранил эти изменения на github. В результате этих действий в нашей локальной директории файл `game` еще хранится, а на github его уже нет, причем на github именно актуальное состояние. Требуется обновить локальную директорию до актуального состояния на github. Для этого достаточно выполнить уже рассмотренную команду `git pull`, как показано на рис. 4.28.

Система `git` очень гибкая и разветвленная, она содержит огромное количество различных команд. Фактически мы рассмотрели лишь базовые возможности. Все подробности можно узнать из литературы, специально посвященной этому вопросу [9].

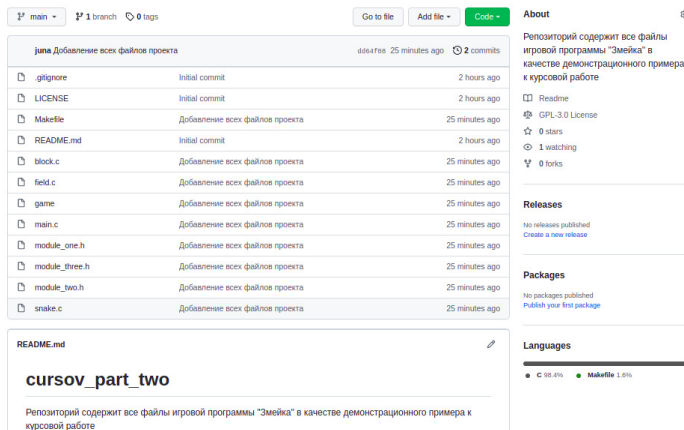


Рис. 4.27. Изменение содержимого репозитория на github

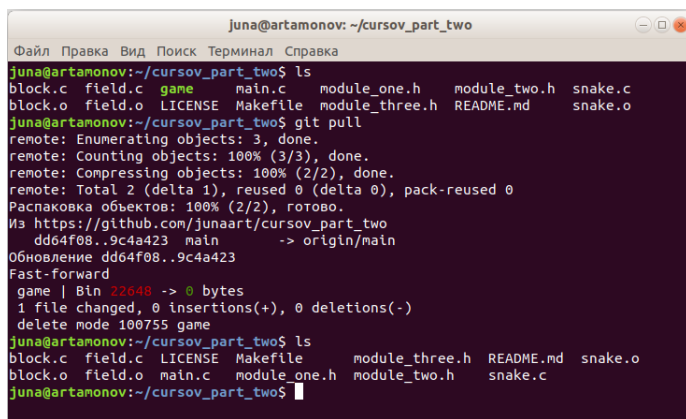


Рис. 4.28. Синхронизация локального репозитория с репозиторием на github

4.3 Задания для самостоятельной работы

Источники: [6, 11].

1. Игра в кости: играющий называет любое число в диапазоне от 2 до 12 и ставку, которую он делает в этот ход. Программа с помощью датчика случайных чисел дважды выбирает числа от 1 до 6 («бросает кубик», на гранях которого цифры от 1 до 6). Если сумма выпавших цифр меньше 7 и играющий задумал число меньше 7, он выигрывает сделанную ставку. Если сумма выпавших цифр больше 7 и играющий задумал число больше 7, он также выигрывает сделанную ставку. Если играющий угадал сумму цифр, он получает в четыре раза больше очков, чем сделанная ставка. Ставка проиграна, если не имеет место ни одна из описанных ситуаций. В начальный момент у играющего 100 очков. Предложите и реализуйте не менее трех вариантов расширения функциональности этой игры (например, 1. вместо человека играет другая программа, 2. казино копит проигранные ставки и т.п.).
2. Игра в морской бой: на поле 10x10 клеток программа размещает флот кораблей: 1 корабль — ряд из 4 клеток («четырёхпалубный»; линкор); 2 корабля — ряд из 3 клеток («трёхпалубные»; крейсера); 3 корабля — ряд из 2 клеток («двухпалубные»; эсминцы); 4 корабля — 1 клетка («однопалубные»; торпедные катера). При размещении корабли не могут касаться друг друга сторонами и углами. Предложите и реализуйте не менее трех вариантов расширения функциональности этой игры (например, играют два человека, играют две программы и т.п.).
3. Программа «Календарь-ежедневник»: на заданную дату в заданное время реализовать возможность запланировать событие. Реализовать функции создания, редактирования,

удаления событий. Предложите и реализуйте не менее трех вариантов расширения функциональности этой программы (можно ориентироваться на функциональность ежедневника Outlook и аналогичных менеджеров).

4. Игра «Жизнь»: имеется поле, поделенное на клетки. Каждая клетка на этой поверхности может находиться в двух состояниях – «живая» или «мёртвая». Распределение «живых» клеток в начале игры называется первым поколением. Каждое следующее поколение (шаг) рассчитывается на основе предыдущего по таким правилам: пустая («мёртвая») клетка с Nx «живыми» клетками-соседями оживает; если у «живой» клетки есть не менее $(Nx - 1)$ «живых» соседей, то эта клетка продолжает жить, если «живых» соседей меньше $(Nx - 1)$ или больше $(Nx + 1)$, то клетка «умирает» (от «одиночества» или от «перенаселённости»). Предложите и реализуйте не менее трех вариантов расширения функциональности этой игры (например, игрок сам устанавливает правила взаимодействия клеток, расставляет «живые» клетки первого поколения, конфигурации генерируются случайно и т.п.).
5. Устный счет: используя арифметические операции умножения, сложения, вычитания, деления, возведения в степень, программа случайно генерирует арифметическое выражение, используя в случае необходимости скобки, вычисляет его значение и предлагает человеку на время также выполнить вычисление. Предложите и реализуйте не менее трех вариантов расширения функциональности этой игры (например, регулируется длина выражения, используемые арифметические операции, предусматривается система бонусов, если игрок угадывает решение и т.п.).
6. Игра «Сапёр»: на поле 10×10 программа случайно размещает фиксированное количество мин. Требуется разминировать поле. Игроку разрешается открывать любую клетку или помечать ее признаком - заминировано. После каж-

дой открытой клетки программа сообщает для каждой смежной с ней клетки, сколько смежных с ней мин. Если игрок открыл клетку с миной, он проиграл. Если он обезвредил все мины, он выиграл. Предложите и реализуйте не менее трех вариантов расширения функциональности этой игры (например, игрок сам выбирает размерность поля, количество мин, количество ошибок и т.п.).

7. Игра «Пятнашки»: в клетках поля 4x4 размещаются фишки от 1 до 15, одна клетка остается пустой. В пустую клетку можно перемещать любую из соседних с ней клеток. Необходимо реализовать перемещение фишек, визуализацию поля. Требуется вернуть искомую нумерацию. Предложите и реализуйте не менее трех вариантов расширения функциональности этой игры (например, программа сама тасует фишки, сама пытается собрать требуемую конфигурацию и т.п.).
8. Игра «Лабиринт»: играющий перемещается в двухмерном пространстве по помещениям здания, план которого играющему неизвестен. Начиная с произвольного помещения, путешественник должен найти выход из здания. Каждое помещение может иметь четыре двери: север, восток, юг, запад и соединяться с другими помещениями. План здания необходимо сгенерировать случайно. Порядок следования помещений в списке должен быть произвольным. Находясь в N-ом помещении, игрок может получить подсказку о правильном направлении движения, если верно выполнит тестовое задание по теме «Программирование на языке высокого уровня». Предложите и реализуйте не менее трех вариантов расширения функциональности этой игры (например, автоматическое или ручное прохождение роботом лабиринта, несколько выходов и т.п.).
9. Программа «Моделирование компьютера»: имеется специальный регистр - аккумулятор, в который помещается результат арифметических операций, различных операций

сравнения. Имеется оперативная память из 100 ячеек, куда можно записать любое целое число. Программа состоит из команд, каждая команда - это четырехзначное число. Первые две цифры - это код команды, которую нужно выполнить: 10 - вводится слово с терминала в указанное место памяти; 11 - выводится слово на терминал из указанного адреса памяти; 20 - в аккумулятор помещается слово из указанного адреса памяти; 21 - в память помещается слово из аккумулятора; 30 - сложение слова аккумулятора и слова из указанного места в памяти (результат остается в аккумуляторе); 31 - вычитание слова аккумулятора и слова из указанного места в памяти (результат остается в аккумуляторе); 32 - деление слова аккумулятора на слово из указанного места памяти; 40 - переход по указанному адресу памяти; 41 - переход по указанному адресу памяти, если в аккумуляторе находится отрицательное число; 42 - переход по указанному адресу памяти, если в аккумуляторе находится нуль; 43 - останов, выполняется при завершении программой своей работы. Необходимо реализовать ввод и выполнение программы. Предложите и реализуйте не менее трех вариантов расширения функциональности этой программы (например, можно расширить набор команд, пытаться программой писать программы самой себе, составить несколько полезных программ т.п.).

10. Крестики-нолики: в любую пустую клетку на поле 3x3 программа может ставить нолик, человек - крестик. Выигрывает тот, кто первым составит горизонтальный, вертикальный или диагональный ряд своих символов (крестиков или ноликов). Предложите и реализуйте не менее трех вариантов расширения функциональности этой игры (например, меняется размер поля, программа следует какой-то стратегии в игре, программы играют друг с другом и т.п.).

Распределение заданий по вариантам

Номер варианта:	1	2	3	4	5	6	7	8	9	10	11
Номер задания:	1	2	3	4	5	6	7	8	9	10	1

Номер варианта:	12	13	14	15	16	17	18	19	20
Номер задания:	2	3	4	5	6	7	8	9	10

Номер варианта:	21	22	23	24	25	26	27	28	29
Номер задания:	9	3	4	5	1	7	8	2	10

ЛИТЕРАТУРА

- [1] Керниган Б., Ритчи Д. Язык программирования Си. – СПб.: «Невский Диалект», 2001. – 352 с.
- [2] Валединский В.Д., Корнев А.А. Методы программирования в примерах и задачах. – М.: Изд-во ЦПИ при механико-математическом ф-те МГУ, 2000. – 152 с.
- [3] Виленкин Н. Метод последовательных приближений. – М.: Наука, 1968. – 108 с.
- [4] Документация по ncurses. Режим доступа <https://tldp.org/HOWTO/NCURSES-Programming-HOWTO/intro.html>, дата обращения 2.01.2022.
- [5] Колисниченко Д. Н. Разработка Linux-приложений. – СПб.: БХВ-Петербург, 2012. – 432 с.
- [6] Пол Дейтел, Харви Дейтел Как программировать на С. – М.: Бином, 2017. – 994 с.
- [7] Савватеев А. Постижение числа π . Режим доступа: https://youtu.be/c1AuZAvPs_s, дата обращения: 28.11.2021.

- [8] *Столяров А.В.* Программирование: введение в профессию. II: Низкоуровневое программирование. – М.: МАКС Пресс, 2016. – 496 с.
- [9] *Чакон С., Штрауб Б.* Git для профессионального программиста. – СПб.: Питер, 2016. – 496 с.
- [10] *Эпштейн М.С.* Практикум по программированию на языке С. – М.: Издательский центр «Академия», 2007. – 128 с.
- [11] *Юркин А.* Задачник по программированию. – СПб.: Питер, 2002. – 192 с.