










Глава I.

Введение в программирование

Глава I. Введение в программирование	1
1. Простейшие программы	3
Зачем нужно программирование?	3
Два этапа создания программ	3
Простейшая программа на Си	4
Вывод текста на экран	4
Как запустить программу?	5
Остановим мгновение	6
2. Переменные	8
Типы данных и переменные	8
Вычисление суммы двух чисел (ввод и вывод)	8
Арифметические выражения	10
Форматы для вывода данных	12
3. Циклы	15
Зачем нужны циклы?	15
Цикл с известным числом шагов (for)	15
Цикл с условием (while)	17
Цикл с постусловием (do — while)	18
Вычисление сумм последовательностей	19
2. Выбор вариантов	22
Зачем нужны условные операторы?	22
Условный оператор if — else	22
Сложные условия	23
Досрочный выход из цикла	25
Переключатель switch (множественный выбор)	26
3. Методы отладки программ	28
Отладочные средства Borland C 3.1	28
Практические приемы	29
4. Работа в графическом режиме	31
Общая структура программы	31
Простейшая графическая программа	31
Графические функции	32
Пример программы	33
5. Процедуры	35
Пример задачи с процедурой	35
Улучшение процедуры	36
6. Функции	38
Отличие функций от процедур	38
Логические функции	39
Функции, возвращающие два значения	40
7. Структура программ	42
Составные части программы	42
Глобальные и локальные переменные	42
Оформление текста программы	43

8.	Анимация	46
	Что такое анимация?	46
	Движение объекта	46
	Отскок от поверхности	48
	Управление клавишами-стрелками	49
9.	Случайные и псевдослучайные числа	52
	Что такое случайные числа?	52
	Распределение случайных чисел	52
	Функции для работы со случайными числами	52
	Случайные числа в заданном интервале	53
	Снег на экране	54

1. Простейшие программы



Зачем нужно программирование?

Иногда создается впечатление, что все существующие задачи могут быть решены с помощью готовых программ для компьютеров. Во многом это действительно так, но опыт показывает, что всегда находятся задачи, которые не решаются (или плохо решаются) стандартными средствами. В этих случаях приходится писать собственную программу, которая делает все так, как вы этого хотите (или нанимать за большие деньги умного дядю, который способен это сделать).



Два этапа создания программ

Программа на языке Си, также как и на большинстве современных языков, создается в два этапа

- **трансляция** программы - перевод текста вашей программы в машинные коды
- **компоновка** - сборка различных частей программы и подключение стандартных библиотек.

Схема создания программы на Си может быть изображена так



Почему же не сделать все за один шаг? Для простейших программ это действительно было бы проще, но для сложных проектов двухступенчатый процесс имеет явные преимущества:

- обычно сложная программа разбивается на несколько отдельных частей (**модулей**), которые отлаживаются отдельно и зачастую разными людьми; поэтому в завершении остается лишь собрать готовые модули в единый проект;
- при исправлении в одном модуле не надо снова транслировать все остальные (это могут быть десятки тысяч строк);
- на компоновке можно подключать модули, написанные на других языках, например, на Ассемблере (в машинных кодах).

Трансляторы языка Си являются **компиляторами**, то есть они переводят (транслируют) сразу всю программу в машинный код, а не транслируют строчка за строчкой во время выполнения, как это делают **интерпретаторы**, например, Бейсик. Это позволяет значительно

ускорить выполнение программы и не ставить интерпретатор на каждый компьютер, где программа будет выполняться.

Исходные файлы программы на языке Си имеют расширение ***.c** или ***.cpp** (если в них использованы специальные возможности Си++ — расширения языка Си). Это обычные текстовые файлы, которые содержат текст программы. Транслятор переводит их в файлы с теми же именами и расширением ***.obj** — это так называемые **объектные файлы**. Они уже содержат машинный код для каждого модуля, но еще не могут выполняться - их надо связать вместе и добавить библиотеки стандартных функций (они имеют расширение ***.lib**). Это делает компоновщик, который умеет также включать в программу файлы ***.obj**, исходный код которых был написан на других языках. В результате получается один файл ***.exe**, который и представляет собой готовую программу.



Простейшая программа на Си

Такая программа состоит всего из 12 символов, но заслуживает внимательного рассмотрения. Вот она:

```
void main()
{
}
```

Основная программа в Си всегда называется именем `main` (будьте внимательны - Си различает большие и маленькие буквы, а все стандартные операторы Си записываются маленькими буквами). Пустые скобки означают, что `main` не имеет аргументов, а слово `void` (пустой) говорит о том, что она также и не возвращает никакого значения, то есть, является **процедурой**. Фигурные скобки обозначают начало и конец процедуры `main` - поскольку внутри них ничего нет, наша программа ничего не делает, она просто соответствует правилам языка Си, ее можно скомпилировать и получить `exe`-файл.



Вывод текста на экран

Составим теперь программу, которая делает что-нибудь полезное, например, выводит на экран слово «Привет».

```
#include <stdio.h>
void main()
{
    printf("Привет");
}
```

подключение функций
стандартного ввода и вывода,
описание которых находится в
файле **stdio.h**

вызов функции вывода на экран



Что новенького?

Перечислим новые элементы, использованные в этой программе:

- Чтобы использовать стандартные функции, необходимо сказать транслятору, что есть функция с таким именем и перечислить тип ее аргументов - тогда он сможет определить, верно ли мы ее используем. Это значит, что надо подключить **описание** этой функции. Описания стандартных функций Си находятся в так называемых **заголовочных файлах** с расширением ***.h** (в каталоге **C:\BORLANDC\INCLUDE**)
- Для подключения заголовочных файлов используется директива (команда) **препроцессора** **"#include"**, после которой в угловых скобках ставится имя файла заголовка (**Преппроцессор**

- это специальная программа, которая обрабатывает текст вашей программы раньше транслятора. Все команды препроцессора начинаются знаком "#"). Внутри угловых скобок не должно быть пробелов. Для подключения еще каждого нового заголовочного файла надо использовать новую команду **"#include"**.

- Для вывода информации на экран используется функция `printf`. В простейшем случае она принимает единственный аргумент - строку в кавычках, которую надо вывести на экран.
- Каждый оператор языка Си заканчивается точкой с запятой.



Как запустить программу?

Чтобы проверить эту программу, надо сначала "напустить" на нее транслятор, который переведет ее в машинные коды, а затем компоновщик, который подключит стандартные функции и создаст исполняемый файл. раньше все это делали, вводя команды в командной строке или с помощью так называемых пакетных файлов. На современном уровне все этапы создания, трансляции, компоновки, отладки и проверки программы объединены и выполняются внутри специальной программы-оболочки, которую называют **интегрированная среда разработки (IDE - integrated development environment)**. В нее входят

- редактор текста
- транслятор
- компоновщик
- отладчик

В этой среде вам достаточно набрать текст программы и нажать на одну клавишу, чтобы она выполнялась (если нет ошибок).

Для запуска оболочки надо набрать в командной строке¹

bc

На экране появляется окно оболочки и, скорее всего, последняя программа, с которой в ней работали. Чтобы закрыть все лишние окна, надо нажимать на клавиши **Alt-F3** до тех пор, пока весь экран не станет серого цвета - это значит, что все окна убраны с рабочего стола.

Теперь нажмите клавишу **F3** и введите имя новой программы (не более 8 символов) и нажмите клавишу **Enter**. Будет открыто новое пустое окно, в котором надо ввести текст программы.

Когда вы набрали всю программу, нажмите клавишу **F2**, чтобы сохранить ее на диске. Рекомендуется делать это чаще на случай отключения питания или сбоя компьютера.

После этого можно запустить программу на выполнение, нажав клавиши **Ctrl-F9**. Если в ней есть ошибки, вы увидите в нижней части экрана окно **Message** (Сообщение), в котором перечислены ошибки (**Error**) и предупреждения (**Warning**), причем та строка, в которой транслятору что-то не понравилось, выделяется голубым цветом в окне программы. Активным сейчас является окно сообщений, если вы сдвигаете в нем курсор (зеленую строку) стрелками вверх и вниз, то в окне программы также сдвигается голубая полоса — вы перешли к другой ошибке и увидели строку, в которой машина ее обнаружила. При поиске ошибок надо помнить, что

- часто реальная ошибка заключена не в выделенной строке, а в предыдущей - проверяйте и ее тоже
- часто одна ошибка вызывает еще несколько, и появляются так называемые наведенные ошибки
- самые распространенные ошибки:

¹ Для специалистов: каталог, в котором расположен файл **bc.exe** должен быть включен в системный путь **PATH**, определяемый в файле **autoexec.bat**.

Unable to open include file 'xxx.h'	не найден заголовочный файл 'xxx.h' (неверно указано его имя, он удален или т.п.)
Function 'xxx' should have a prototype	функция 'xxx' не объявлена (не подключен заголовочный файл или не объявлена своя функция, или неверное имя функции)
Unterminated string or character constant	не закрыты кавычки
Statement missing ;	нет точки с запятой в конце оператора
Compound statement missing }	не закрыта фигурная скобка
Undefined symbol 'xxx'	не объявлена переменная 'xxx'



Основные клавиши оболочки *Borland C*

F1	помощь (справочная система)
F2	запись файла в активном окне на диск
F3	чтение файла с диска или создание нового файла
Alt-F3	закрыть активное окно
F5	раскрыть окно на весь экран
Ctrl-F9	запуск программы на выполнение
Alt-F5	показать рабочий экран
Alt-X	выход из оболочки



Работа с блоками текста

Часто надо скопировать, удалить или переместить часть текста. Для этого есть специальные клавиши работы с блоками. Выделенный блок помечается в редакторе серым цветом.

Ctrl-Y	удалить строку, в которой стоит курсор
Shift-ctrl-Left	расширить выделенный блок
Ctrl-Del, Shift-Del	удалить выделенный блок
Ctrl-Insert	запомнить выделенный блок в буфере
Shift-Insert	вставить блок из буфера
Ctrl-K + B	установить начало блока
Ctrl-K + K	установить конец блока
Ctrl-K + L	выделить текущую строку (в которой стоит курсор)
Ctrl-K + C	скопировать выделенный блок выше курсора
Ctrl-K + V	переместить выделенный блок выше курсора
Ctrl-K + H	отменить выделение блока



Остановим мгновение

Если запускать эту программу из оболочки *Borland C*, то обнаружится, что программа сразу заканчивает работу и возвращается обратно в оболочку, не дав посмотреть результат ее

работы на экране. Борьбаться с этим можно так - давайте скажем компьютеру, что в конце работы надо дождаться нажатия любой клавиши.

```
#include <stdio.h>
#include <conio.h>
void main()
{
    printf("Привет");
    getch();
}
```

подключение функций консольного ввода и вывода, описание которых находится в файле *conio.h*

// вывод на экран
/*ждать нажатия клавиши*/

вызов функции, которая ждет нажатия на любую клавишу



Что новенького?

- Задержка до нажатия любой клавиши выполняется функцией `getch()`.
- Описание этой функции находится в заголовочном файле *conio.h*.
- Знаки `//` обозначают начало **комментария** — все правее них до конца строки не обрабатывается компьютером и служит нам для пояснения программы.
- Комментарий также можно ограничивать парами символов `/*` (начало комментария) и `*/` (конец комментария). В этом случае комментарий может быть многострочный, то есть состоять из нескольких строк.

2. Переменные



Типы данных и переменные

Для обработки данных их необходимо хранить в памяти. При этом к этим данным надо как-то обращаться. Обычно люди обращаются друг к другу по имени, такой же способ используется в программировании: каждой ячейке памяти (или группе ячеек) дается свое собственное имя. Используя это имя можно прочитать информацию из ячейки и записать туда новую информацию.

Переменная - это ячейка в памяти компьютера, которая имеет имя и хранит некоторое значение. Значение переменной может меняться во время выполнения программы. При записи в ячейку нового значения старое стирается.

С точки зрения компьютера все данные в памяти - это числа (более точно - наборы нулей и единиц). Тем не менее, и вы (и компьютер) знаете, что с целыми и дробными числами работают по-разному. Поэтому в каждом языке программирования есть разные типы данных (переменных), для обработки которых используются разные методы. Основными данными в языке Си являются

- целые переменные (тип `int` - от английского *integer* - целый) занимают 2 байта в памяти
- вещественные переменные, которые могут иметь дробную часть (тип `float` - от английского *floating point* - плавающая точка), занимают 4 байта в памяти
- символы (тип `char` - от английского *character* - символ) занимают 1 байт в памяти

Для использования все переменные необходимо объявлять - то есть сказать компьютеру, чтобы он выделил на них ячейку памяти нужного размера и присвоил ей нужное имя. Переменные обычно объявляются в начале программы. Для объявления надо написать название типа переменных (`int`, `float` или `char`), а затем через запятую имена всех объявляемых переменных. При желании можно сразу записать в новую ячейку нужное число, как показано в примерах ниже. Если переменной не присваивается никакого значения, то в ней находится "мусор", то есть то, что было там раньше.

Примеры.

```
int a;                // выделить память под целую
                     // переменную a

float b, c;           // две вещественных переменных b и c

int Tu104, Il86=23, Yak42; // три целых переменных, причем в Il86
                          // сразу записывается число 23

float x=4.56, y, z;   // три вещественных переменных,
                     // причем в x сразу записывается число 4.56

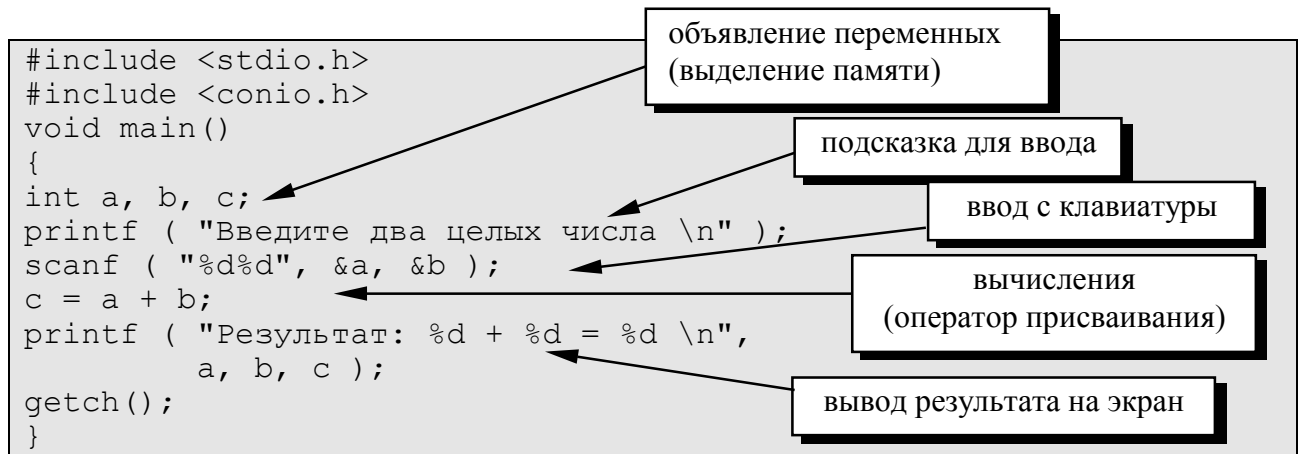
char c, c2='A', m;    // три символьных переменных, причем в c2
                     // сразу записывается символ 'A'
```



Вычисление суммы двух чисел (ввод и вывод)

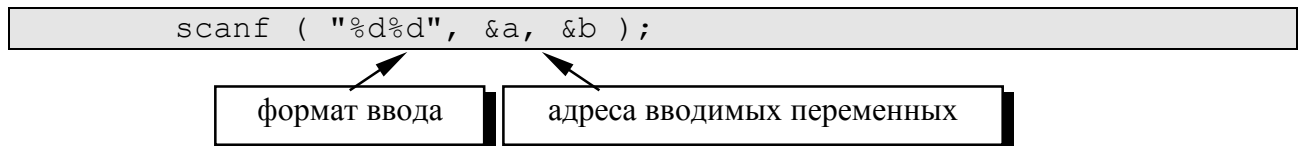
Задача. Ввести с клавиатуры два целых числа и вывести на экран их сумму.

Сразу запишем решение задачи на языке Си.



Что новенького?

- Программа чаще всего содержит 4 части:
 - ⇒ объявление переменных
 - ⇒ ввод исходных данных
 - ⇒ обработка данных (вычисления)
 - ⇒ вывод результата на экран или на печать
- Перед вводом данных необходимо вывести на экран подсказку (иначе компьютер будет ждать ввода данных, а пользователь не будет знать, что от него хочет машина).
- Символы `"\n"` в функции `printf` обозначают переход в начало новой строки.
- Для ввода данных используют функцию `scanf`.



⇒ Формат ввода - это строка в кавычках, в которой перечислены один или несколько форматов ввода данных:

%d	ввод целого числа (переменная типа <code>int</code>)
%f	ввод вещественного числа (переменная типа <code>float</code>)
%c	ввод одного символа (переменная типа <code>char</code>)

⇒ После формата ввода через запятую перечисляются **адреса** ячеек памяти, в которые надо записать введенные значения. Почувствуйте разницу:

a	значение переменной a
&a	адрес переменной a

⇒ Количество форматов в строке должно быть равно количеству адресов в списке. Кроме того, тип переменных должен совпадать с указанным: например, если **a** и **b** - целые переменные, то следующие вызовы функций ошибочны

```
scanf ( "%d%d", &a );
```

неясно, куда записывать второе введенное число

```
scanf ( "%d%d", &a, &b, &c );
```

переменная **c** не будет введена, так как на нее не задан формат

```
scanf ( "%f%f", &a, &b );
```

нельзя вводить целые переменные по вещественному формату

- Для вычислений используют **оператор присваивания**, в котором
 ⇒ справа от знака равенства стоит арифметическое выражение, которое надо вычислить
 ⇒ слева от знака равенства ставится имя переменной, в которую надо записать результат

`c = a + b;` // вычислить сумму **a** и **b** и записать результат в **c**

- Для вывода чисел и значений переменных на экран используют функцию `printf`

```
printf ( "Результат: %d + %d = %d \n", a, b, c );
```

эти символы
просто напечатать

здесь вывести
целые числа

список
переменных

содержание скобок при вызове функции `printf` очень похоже на функцию `scanf`

⇒ Сначала идет символьная строка — формат вывода — в которой можно использовать специальные символы

%d вывод целого числа
%f вывод вещественного числа
%c вывод одного символа
%s вывод символьной строки
\n переход в начало новой строки

все остальные символы (кроме некоторых других специальных команд) просто выводятся на экран.

⇒ В символьной строке мы сказали компьютеру, *какие* данные (целые, вещественные или символьные) надо вывести на экран, но не сказали *откуда* (из каких ячеек памяти) их брать. Поэтому через запятую после формата вывода надо поставить список чисел или переменных, значения которых надо вывести, при этом можно сразу проводить вычисления.

```
printf ( "Результат: %d + %d = %d \n", a, 5, a+5 );
```

⇒ Так же, как и для функции `scanf`, надо следить за совпадением типов и количества переменных и форматов вывода.



Арифметические выражения



Из чего состоят арифметические выражения?

Арифметические выражения, стоящие в правой части оператора присваивания, могут содержать

- целые и вещественные числа (в вещественных числах целая и дробная часть разделяются точкой, а не запятой, как это принято в математике)
- знаки арифметических действий

+ **—** сложение, вычитание
***** **/** умножение, деление
% остаток от деления

- вызовы стандартных функций

`abs (i)` модуль целого числа **i**

<code>fabs (x)</code>	модель вещественного числа x
<code>sqrt (x)</code>	квадратный корень из вещественного числа x
<code>pow (x, y)</code>	вычисляет x в степени y

- круглые скобки

Особенности арифметических операций

При использовании деления надо помнить, что

При делении целого числа на целое остаток от деления отбрасывается, таким образом, $7/4$ будет равно 1. Если же надо получить вещественное число и не отбрасывать остаток, делимое или делитель надо преобразовать к вещественной форме. Например:

```
int i, n;    float x;
i = 7;
x = i / 4;    // x=1, делится целое на целое
x = i / 4.;   // x=1.75, делится целое на дробное
x = (float) i / 4; // x=1.75, делится дробное на целое
n = 7. / 4.;   // n=1, результат записывается в
               // целую переменную
```

Наибольшие сложности из всех действий вызывает взятие остатка. Если надо вычислить остаток от деления переменной **a** на переменную **b** и результат записать в переменную **ostatok**, то оператор присваивания выглядит так:

```
ostatok = a % b;
```

Приоритет арифметических операций

В языках программирования арифметические выражения записываются в одну строчку, поэтому необходимо знать **приоритет (старшинство) операций**, то есть последовательность их выполнения. Сначала выполняются

- операции в скобках, затем
- вызовы функций, затем
- умножение, деление и остаток от деления, слева направо, затем
- сложение и вычитание, слева направо.

Например:

	<div style="border: 1px solid black; padding: 2px; display: inline-block;">2</div>	<div style="border: 1px solid black; padding: 2px; display: inline-block;">1</div>	<div style="border: 1px solid black; padding: 2px; display: inline-block;">5</div>	<div style="border: 1px solid black; padding: 2px; display: inline-block;">4</div>	<div style="border: 1px solid black; padding: 2px; display: inline-block;">3</div>	<div style="border: 1px solid black; padding: 2px; display: inline-block;">8</div>	<div style="border: 1px solid black; padding: 2px; display: inline-block;">6</div>	<div style="border: 1px solid black; padding: 2px; display: inline-block;">7</div>
	↓	↓	↓	↓	↓	↓	↓	↓
x = (a + 5 * b) * fabs (c + d) - (3 * b - c);								

Для изменения порядка выполнения операций используются круглые скобки. Например, выражение

$$y = \frac{4x + 5}{(2x - 15z)(3z - 3)} - \frac{5x}{x + z + 3}$$

в компьютерном виде запишется в виде

```
y = (4*x + 5) / ((2*x - 15*z) * (3*z - 3)) - 5 * x /
      (x + z + 3);
```



Странные операторы присваивания

В программировании часто используются несколько странные операторы присваивания, например:

```
i = i + 1;
```

Если считать это уравнением, то оно бессмысленно с точки зрения математики. Однако с точки зрения информатики этот оператор служит для увеличения значения переменной *i* на единицу. Буквально это означает: взять старое значение переменной *i*, прибавить к нему единицу и записать результат в ту же переменную *i*.



Инкремент и декремент

В языке Си определены специальные операторы быстрого увеличения на единицу (*инкремента*)

```
i ++;      // или
++ i;
```

что равносильно оператору присваивания

```
i = i + 1;
```

и быстрого уменьшения на единицу (*декремента*)

```
i —;      // или
— i;
```

что равносильно оператору присваивания

```
i = i - 1;
```

Между первой и второй формами этих операторов есть некоторая разница, но только тогда, когда они входят в состав более сложных операторов или условий.



Сокращенная запись арифметических выражений

Если мы хотим изменить значение какой-то переменной (взять ее старое значение, что-то с ним сделать и записать результат в эту же переменную), то удобно использовать сокращенную запись арифметических выражений:

Сокращенная запись	Полная запись
<code>x += a;</code>	<code>x = x + a;</code>
<code>x -= a;</code>	<code>x = x - a;</code>
<code>x *= a;</code>	<code>x = x * a;</code>
<code>x /= a;</code>	<code>x = x / a;</code>
<code>x %= a;</code>	<code>x = x % a;</code>



Форматы для вывода данных



Целые числа

Первым параметром при вызове функций `scanf` и `printf` должна стоять символьная строка, определяющая формат ввода или данных. Для функции `scanf`, которая выполняет ввод данных, достаточно просто указать один из форматов `"%d"`, `"%f"` или `"%c"` для ввода целого числа, вещественного числа или символа, соответственно. В то же время форматная

строка в функции `printf` позволяет управлять выводом на экран, а именно, задать размер поля, которое отводится для данного числа.

Ниже показаны примеры форматирования при выводе целого числа 1234. Чтобы увидеть поле, которое отводится для числа, оно ограничено слева и справа скобками.

Пример вывода	Результат	Комментарий
<code>printf ("[%d]", 1234);</code>	[1234]	На число отводится минимально возможное число позиций.
<code>printf ("[%6d]", 1234);</code>	[1234]	На число отводится 6 позиций, выравнивание вправо.
<code>printf ("[% -6d]", 1234);</code>	[1234]	На число отводится 6 позиций, выравнивание влево.
<code>printf ("[%2d]", 1234);</code>	[1234]	Число не помещается в заданные 2 позиции, поэтому область вывода расширяется.

Для вывода символов используются такие же приемы форматирования, но формат `"%d"` заменяется на `"%c"`.



Вещественные числа

Для вывода (и для ввода) вещественных чисел могут использоваться три формата: `"%f"`, `"%e"` и `"%g"`. В таблице показаны примеры использования формата `"%f"`.

Пример вывода	Результат	Комментарий
<code>printf ("[%f]", 123.45);</code>	[123.450000]	На число отводится минимально возможное число позиций, выводится 6 знаков в дробной части.
<code>printf (" [%9.3f]", 123.45);</code>		На число отводится всего 9 позиций, из них 3 – для дробной части, выравнивание вправо.
<code>printf (" [% -9.3f]", 123.45);</code>	[123.450]	На число отводится всего 9 позиций, из них 3 – для дробной части, выравнивание влево.
<code>printf (" [%6.4f]", 123.45);</code>	[123.4500]	Число не помещается в заданные 6 позиций (4 цифры в дробной части), поэтому область вывода расширяется.

Формат `"%e"` применяется в научных расчетах для вывода очень больших или очень маленьких чисел, например, размера атома или расстояния до Солнца. С представлением числа в так называемом *стандартном виде* (с выделенной *мантиссой* и *порядком*). Например, число 123.45 может быть записано в стандартном виде как $123.45 = 1.2345 \times 10^2$. Здесь 1.2345 – мантисса (она всегда находится в интервале от 1 до 10), а 2 – порядок (мантисса умножается на 10 в этой степени). При выводе по формату `"%e"` также можно задать число позиций, которые отводятся для вывода числа, и число цифр в дробной части мантиссы. Порядок всегда указывается в виде двух цифр, перед которыми стоит буква `"e"` и знак порядка (плюс или минус).

Пример вывода	Результат	Комментарий
<code>printf ("[%e]", 123.45);</code>	<code>[1.234500e+02]</code>	На число отводится минимально возможное число позиций, выводится 6 знаков в дробной части мантииссы.
<code>printf("[%12.3e]", 123.45);</code>	<code>[1.234e+02]</code>	На число отводится всего 12 позиций, из них 3 – для дробной части мантииссы, выравнивание вправо.
<code>printf("[%-12.3e]", 123.45);</code>	<code>[1.234e+02]</code>	На число отводится всего 12 позиций, из них 3 – для дробной части мантииссы, выравнивание влево.
<code>printf("[%6.2e]", 123.45);</code>	<code>[1.23e+02]</code>	Число не помещается в заданные 6 позиций (2 цифры в дробной части мантииссы), поэтому область вывода расширяется.

Формат "%g" применяется для того, чтобы удалить лишние нули в конце дробной части числа и автоматически выбрать формат (в стандартном виде или с фиксированной точкой). Для очень больших или очень маленьких чисел выбирается формат с плавающей точкой (в стандартном виде). В этом формате можно задать общее число позиций на число и количество значащих цифр.

Пример вывода	Результат	Комментарий
<code>printf ("%g", 12345);</code> <code>printf ("%g", 123.45);</code> <code>printf ("%g", 0.000012345);</code>	<code>[12345]</code> <code>[123.45]</code> <code>[1.2345e-05]</code>	На число отводится минимально возможное число позиций, выводится не более 6 значащих цифр.
<code>printf ("%10.3g", 12345);</code> <code>printf ("%10.3g", 123.45);</code> <code>printf ("%10.3g", 0.000012345);</code>	<code>[1.23e+04]</code> <code>[123]</code> <code>[1.23e-05]</code>	На число отводится всего 10 позиций, из них 3 значащие цифры, выравнивание вправо. Чтобы сделать выравнивание влево, используют формат "%-10.3g".

3. Циклы

Зачем нужны циклы?

Теперь посмотрим, как вывести на экран это самое приветствие 10 раз. Конечно, можно написать 10 раз оператор `printf`, но если надо вывести строку 200 раз, то программа значительно увеличится. Поэтому надо использовать **циклы**.

Цикл - это последовательность команд, которая выполняется несколько раз.

В языке Си существует несколько видов циклов.

Цикл с известным числом шагов (`for`)

Часто мы заранее знаем (или можем рассчитать), сколько раз нам надо выполнить какую-то операцию. В некоторых языках программирования для этого используется цикл `repeat` - повтори заданное количество раз. Подумаем, как выполнять такой цикл. В самом деле, в памяти выделяется ячейка и в нее записывается число повторений. Когда программа выполняет тело цикла один раз, содержимое этой ячейки (**счетчик**) уменьшается на единицу. Выполнение цикла заканчивается, когда в этой ячейке будет нуль.

В языке Си цикла `repeat` нет, а есть цикл `for`. Он не скрывает ячейку-счетчик, а требует явно объявить ее (выделить под нее память), и даже позволяет использовать ее значение в теле цикла. Ниже показан пример программы, которая печатает приветствие 10 раз.

<pre>#include <stdio.h> #include <conio.h> void main() { int i; for (i=1; i <= 10; i ++) { printf("Привет"); } getch(); }</pre>	объявление переменной i (выделение памяти)
	заголовок цикла
	тело цикла

Что новенького?

- Цикл `for` используется тогда, когда количество повторений цикла заранее известно или может быть вычислено.
- Цикл `for` состоит из заголовка и тела цикла.
- В заголовке после слова `for` в круглых скобках записываются через точку с запятой три выражения:

⇒ начальные условия: операторы присваивания, которые выполняются один раз перед выполнением цикла

⇒ условие, при котором выполняется следующий шаг цикла; если условие неверно, работа цикла заканчивается; если оно неверно в самом начале, цикл не выполняется ни

одного раза (говорят, что это - *цикл с предусловием*, то есть условие проверяется перед выполнением цикла);

⇒ действия в конце каждого шага цикла (в большинстве случаев это операторы присваивания)

- В каждой части заголовка может быть несколько операторов, разделенных запятыми. Примеры заголовков:

```
for ( i = 0; i < 10; i ++ ) { ... }
for ( i = 0, x = 1.; i < 10; i += 2, x *= 0.1 ){ ... }
```

- Тело цикла заключается в фигурные скобки; если в теле цикла стоит всего один оператор, скобки можно не ставить.
- В тело цикла могут входить любые другие операторы, в том числе и другие циклы (такой прием называется "*вложенные циклы*").
- Для того, чтобы легче разобраться в программе, все тело цикла и ограничивающие его скобки сдвигаются вправо на 2-3 символа.



Чему равен квадрат числа?

Напишем программу, которая вводит с клавиатуры натуральное число N и выводит на экран квадраты всех целых чисел от 1 до N таком виде

Квадрат числа 1 равен 1

Квадрат числа 2 равен 4

...

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int i, N;                                // i - переменная цикла
    printf ( "Введите число N: " ); // подсказка для ввода
    scanf ( "%d", &N );                // ввод N с клавиатуры
    for ( i = 1; i <= N; i ++ )          // цикл: для всех i от 1 до N
    {
        printf ( "Квадрат числа %d равен %d\n", i, i*i);
    }
    getch();
}
```

Мы объявили две переменные: N — максимальное число, i — вспомогательная переменная, которая в цикле принимает последовательно все значения от 1 до N . Для ввода значения N мы напечатали на экране подсказку и использовали функцию `scanf` с форматом `%d` (ввод целого числа).

При входе в цикл выполняется оператор $i = 1$, и затем переменная i с каждым шагом увеличивается на 1 ($i ++$). Цикл выполняется пока истинно условие $i <= N$. В теле цикла единственный оператор вывода печатает на экране само число и его квадрат по заданному формату. Для возведения в квадрат или другую невысокую степень лучше использовать умножение.

Цикл с условием (while)

Очень часто заранее невозможно сказать, сколько раз надо выполнить какую-то операцию, но можно определить условие, при котором она должна заканчиваться. Такое задание на русском языке может выглядеть так: делай эту работу до тех пор, **пока** она не будет закончена (пили бревно, пока оно не будет распилено; иди вперед, пока не дойдешь до двери). Слово **пока** на английском языке записывается как `while`, и так же называется еще один вид цикла.

Задача. Ввести целое число (меньше 2 000 000 000) и определить, сколько в нем цифр.

Для решения этой задачи обычно применяется такой алгоритм. Число делится на 10 и отбрасывается остаток, и так до тех пор, пока результат деления не равен нулю. С помощью специальной переменной (она называется *счетчиком*) считаем, сколько раз выполнялось деление - столько цифр и было в числе. Понятно, что нельзя заранее определить, сколько раз надо разделить число, поэтому надо использовать цикл с условием.

В этой задаче еще важно, что число **N** может быть очень большое. Дело в том, что в переменную типа `int` «помещается» целое число в интервале от **-32768** до **32767**. Для больших чисел надо использовать тип `long int` (длинное целое). Переменные этого типа занимают в памяти 4 байта и могут хранить значения в пределах 2 млрд. (примерно). Для ввода и вывода таких чисел используют формат **%ld**.

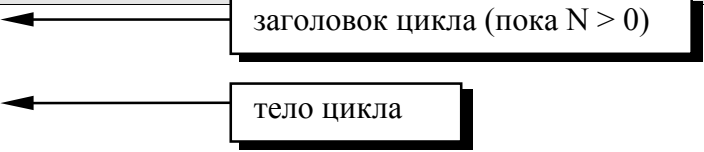
```
#include <stdio.h>
#include <conio.h>

void main()
{
    int count=0; // count - переменная-счетчик
    long N;

    printf ( "\nВведите число N: " ); // подсказка
    scanf ( "%ld", &N ); // ввод N с клавиатуры

    while ( N > 0 )
    {
        N /= 10;
        count ++;
    }

    printf ( "В этом числе %d цифр\n", count );
    getch();
}
```



Что новенького?

- Цикл `while` используется тогда, когда количество повторений цикла заранее неизвестно и не может быть вычислено.
- Цикл `while` состоит из заголовка и тела цикла.
- В заголовке после слова `while` в круглых скобках записывается условие, при котором цикл продолжает выполняться; когда условие становится неверно, цикл заканчивается.
- В условии можно использовать знаки логических отношений и операций
> < больше, меньше

>=	<=	больше или равно, меньше или равно
==		равно
!=		не равно

- Если условие неверно в самом начале, то цикл не выполняется ни разу (это цикл с *предусловием*).
- Если условие никогда не становится *ложным* (неверным), то цикл никогда не заканчивается; в таком случае говорят, что программа "*зациклилась*" — это серьезная логическая ошибка.
- В языке Си любое число, не равное нулю, обозначает истинное условие, а ноль — ложное условие:

```
while ( 1 ){ ... } // бесконечный цикл  
while ( 0 ){ ... } // цикл не выполнится ни разу
```

- Тело цикла заключается в фигурные скобки; если в теле цикла стоит всего один оператор, скобки можно не ставить.
- В тело цикла могут входить любые другие операторы, в том числе и другие циклы (такой прием называется "*вложенные циклы*").
- Для того, чтобы легче разобраться в программе, все тело цикла и ограничивающие его скобки сдвигаются вправо на 3-4 символа.
- Если надо работать с большими целыми числами, используется тип данных `long int`. Для ввода и вывода переменных этого типа используют формат `%ld`.



Цикл с постусловием (`do — while`)

Существуют также случаи, когда надо выполнить цикл хотя бы один раз, а затем на каждом шагу делать проверку некоторого условия и закончить цикл, когда это условие станет ложным. Для этого используется *цикл с постусловием* (то есть условие проверяется не в начале, а в конце цикла). Не рекомендуется применять его слишком часто, поскольку он напоминает такую ситуацию: прыгнул в бассейн, и только потом посмотрел, есть ли в нем вода. Рассмотрим случай, когда его использование оправдано.

Задача. Ввести натуральное число и найти сумму его цифр. Организовать ввод числа так, чтобы нельзя было ввести отрицательное число или ноль.

Любая программа должна обеспечивать защиту от неверного ввода данных (иногда такую защиту называют "защитой от дурака" — *fool proof*). Поскольку пользователь может вводить данные неверно сколько угодно раз, то надо использовать цикл с условием. С другой стороны, по крайней мере один раз надо ввести число, поэтому нужен цикл с постусловием.

В отличие от предыдущей программы, теперь надо при каждом делении определять остаток (последняя цифра числа равна остатку от деления его на 10) и суммировать все остатки в специальной переменной.

```

#include <stdio.h>
#include <conio.h>
void main()
{
    int sum=0, N;      // sum - сумма цифр числа

    do {
        printf ( "\nВведите натуральное число:" );
        scanf ( "%d", &N );
    }
    while ( N <= 0 );

    while ( N > 0 ) {
        sum += N % 10;
        N /= 10;
        count ++;
    }
    printf ( "Сумма цифр этого числа равна %d\n", sum );
    getch();
}

```

Diagram labels and arrows:

- заголовок цикла** (cycle header) points to the `do {` line.
- тело цикла** (cycle body) points to the `printf` and `scanf` lines inside the `do` block.
- условие цикла (пока N <= 0)** (cycle condition) points to the `while (N <= 0);` line.



Что новенького?

- Цикл `do-while` используется тогда, когда количество повторений цикла заранее неизвестно и не может быть вычислено.
- Цикл состоит из заголовка `do`, тела цикла и завершающего условия.
- Условие записывается в круглых скобках после слова `while`, цикл продолжает выполняться, пока условие верно; когда условие становится неверно, цикл заканчивается.
- Условие проверяется только в конце очередного шага цикла (это цикл *с постусловием*), таким образом, цикл всегда выполняется хотя бы один раз.
- Если условие никогда не становится ложным (неверным), то цикл никогда не заканчивается; в таком случае говорят, что программа **"зациклилась"** — это серьезная логическая ошибка.
- Тело цикла заключается в фигурные скобки; если в теле цикла стоит всего один оператор, скобки можно не ставить.
- В тело цикла могут входить любые другие операторы, в том числе и другие циклы (такой прием называется **"вложенные циклы"**).
- Для того, чтобы легче разобраться в программе, все тело цикла и ограничивающие его скобки сдвигаются вправо на 2-3 символа.



Вычисление сумм последовательностей



Суммы с заданным числом элементов

Задача. Найти сумму первых 20 элементов последовательности

$$S = \frac{1}{2} - \frac{2}{4} + \frac{3}{6} - \frac{4}{12} + \dots$$

Чтобы решить эту задачу, надо определить закономерность в изменении элементов. В данном случае можно заметить:

- Каждый элемент представляет собой дробь.
- Числитель дроби при переходе к следующему элементу возрастает на единицу.
- Знаменатель дроби с каждым шагом увеличивается в 2 раза.
- Знаки перед дробями чередуются (плюс, минус и т.д.).

Любой элемент последовательности можно представить в виде

$$a_i = \frac{zc}{d},$$

где изменение переменных **z**, **c** и **d** описываются следующей таблицей (для первых пяти элементов)

i	1	2	3	4	5
z	1	-1	1	-1	1
c	1	2	3	4	5
d	2	3	6	12	24

У переменной **z** меняется знак (эту операцию можно записать в виде **z=-z**), значение переменной **c** увеличивается на единицу (**c++**), а переменная **d** умножается на 2 (**d=d*2**). Алгоритм решения задачи можно записать в виде следующих шагов

- Записать в переменную **S** значение 0. В этой ячейке будет накапливаться сумма.
- Записать в переменные **z**, **c** и **d** начальные значения (для первого элемента): **z=1**, **c=1**, **d=2**.
- Сделать 20 раз:
 - ⇒ добавить к сумме значение очередного элемента
 - ⇒ изменить значения переменных **z**, **c** и **d** для следующего элемента.

```
#include <stdio.h>
void main()
{
    float S, z, c, d;
    int i;

    S = 0; z = 1; c = 1; d = 2;
    for ( i = 1; i <= 20; i ++ )
    {
        S = S + z*c/d;
        z = - z;
        c ++;
        d = d * 2;
    }

    printf("Сумма S = %f", S);
}
```

начальные значения

добавить элемент к сумме

изменить переменные



Суммы с ограничивающим условием

Рассмотрим более сложную задачу, когда количество элементов заранее неизвестно.

Задача. Найти сумму всех элементов последовательности

$$S = \frac{1}{2} - \frac{2}{4} + \frac{3}{6} - \frac{4}{12} + \dots,$$

которые по модулю меньше, чем 0.001.

Эта задача имеет решение только тогда, когда элементы последовательности убывают по модулю и стремятся к нулю. Поскольку мы не знаем, сколько элементов войдет в сумму, надо использовать цикл **while** (или **do - while**). Один из вариантов решения показан ниже.

```
#include <stdio.h>
void main()
{
float S, z, c, d, a;
```

начальные значения

```
S = 0; z = 1; c = 1; d = 2;
```

```
a = 1;
```

любое число больше 0.001

```
while ( a >= 0.001 )
```

```
{
```

```
    a = c / d;
```

вычислить модуль элемента

```
    S = S + z*a;
```

добавить элемент к сумме

```
    z = - z;
```

```
    c ++;
```

изменить переменные

```
    d = d * 2;
```

```
}
```

```
printf("Сумма S = %f", S);
```

```
}
```

Цикл закончится тогда, когда переменная **a** (она обозначает модуль очередного элемента последовательности) станет меньше 0.001. Чтобы программа вошла в цикл на первом шаге, в эту переменную надо записать любое значение, большее, чем 0.001.

Очевидно, что если переменная **a** не будет уменьшаться, то условие в заголовке цикла всегда будет истинно и программа «зациклится».

4. Выбор вариантов



Зачем нужны условные операторы?

В простейших программах все команды выполняются одна за другой последовательно. Однако часто надо выбрать тот или иной вариант действий в зависимости от некоторых условий, то есть, если условие верно, поступать одним способом, а если неверно — другим. Для этого применяют условные операторы. В языке Си существует два вида условных операторов:

- оператор `if-else` для выбора из двух вариантов
- оператор множественного выбора `switch` для выбора из нескольких вариантов



Условный оператор `if — else`

Задача. Ввести с клавиатуры два вещественных числа и определить наибольшее из них.

По условию задачи нам надо вывести один из двух вариантов ответа: если первое число больше второго, то вывести на экран его, если нет — то второе число. ниже показаны два варианта решения этой задачи: в первом результат сразу выводится на экран, а во втором сначала наибольшее из двух чисел записывается в третью переменную.

```
#include <stdio.h>
#include <conio.h>
void main()
{
    float A, B;
    printf ("Введите A и B :");
    scanf ( "%f%f", &A, &B );
```

```
    if ( A > B )
    {
        printf ( "Наибольшее %f",
                A );
    }
    else
    {
        printf ( "Наибольшее %f",
                B );
    }
```

```
    getch();
}
```

```
#include <stdio.h>
#include <conio.h>
void main()
{
    float A, B, Max;
    printf("Введите A и B : ");
    scanf ( "%f%f", &A, &B );
```

```
    if ( A > B )
    {
        Max = A;
```

← заголовок с условием

```
    }
    else
```

← блок "если"

```
    {
        Max = B;
```

← блок "иначе"

```
    }
    printf ( "Наибольшее %f",
            Max );
```

```
    getch();
}
```



Что новенького?

- Условный оператор имеет следующий вид:

```
if ( условие )
{
    ...    // блок "если" — операторы, которые выполняются,
          // если условие в скобках истинно
}
else
{
    ...    // блок "иначе " — операторы, которые выполняются,
          // если условие в скобках ложно
}
```

- Эта запись представляет собой единый оператор, поэтому между скобкой, завершающей блок **"если"** и словом `else` не могут находиться никакие операторы.
- После слова `else` никогда НЕ ставится условие — блок **"иначе"** выполняется тогда, когда основное условие, указанное в скобках после `if`, ложно.
- Если в блоке **"если"** или в блоке **"иначе"** только один оператор, то фигурные скобки можно не ставить.
- В условии можно использовать любые отношения и логические операции.
- Если в блоке **"иначе"** не надо ничего делать (например: **"если в продаже есть мороженое, купи мороженое"**, а если нет ...), то весь блок **"иначе"** можно опустить и использовать сокращенную форму условного оператора:

```
if ( условие )
{
    ...    // блок "если" — операторы, которые выполняются,
           // если условие в скобках истинно
}
```

Например, решение предыдущей задачи могло бы выглядеть так:

```
#include <stdio.h>
#include <conio.h>
void main()
{
    float A, B, Max;
    printf("Введите A и B : ");
    scanf ( "%f%f", &A, &B );
    Max = A;
    if ( B > A )
        Max = B;
    printf ( "Наибольшее %f", Max );
    getch();
}
```

- В блоки **"если"** и **"иначе"** могут входить любые другие операторы, в том числе и другие вложенные условные операторы; при этом оператор `else` относится к ближайшему предыдущему `if`:

```
if ( A > 10 )
{
    if ( A > 100 )
        printf ( "У вас очень много денег." );
    else
        printf ( "У вас достаточно денег." );
}
else
    printf ( "У вас маловато денег." );
```

- Чтобы легче разобраться в программе, все блоки **"если"** и **"иначе"** (вместе с ограничивающими их скобками) сдвигаются вправо на 2-3 символа.



Сложные условия

Простейшие условия состоят из одного отношения (больше, меньше и т.д.). Иногда надо написать условие, в котором объединяются два или более простейших отношений. Например,

фирма отбирает сотрудников возраста 25—40 лет. Тогда простейшая программа могла бы выглядеть так:

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

void main()
{
    int age;
    printf ( "\nВведите ваш возраст: " );
    scanf ( "%d", &age );
    if ( 25 <= age && age <= 40 )
        printf ( "Вы нам подходите." );
    else
        printf ( "Извините, Вы нам не подходите." );
    getch();
}
```

сложное условие



Что новенького?

- Сложное условие состоит из двух или нескольких простых отношений, которые объединяются с помощью знаков *логических операций*:

⇒ операция **"И"** — требуется одновременное выполнение двух условий

условие_1 && условие_2

Эту операцию можно описать следующей таблицей (она называется *таблицей истинности*)

условие 1	условие 2	условие 1 && условие 1
ложно (0)	ложно (0)	ложно(0)
ложно (0)	истинно (1)	ложно(0)
истинно (1)	ложно (0)	ложно(0)
истинно (1)	истинно (1)	истинно (1)

⇒ операция **"ИЛИ"** — требуется выполнение хотя бы одного из двух условий (или обоих сразу)

условие_1 || условие_2

Таблица истинности запишется в виде

условие 1	условие 2	условие 1 условие 1
ложно (0)	ложно (0)	ложно(0)
ложно (0)	истинно (1)	истинно (1)
истинно (1)	ложно (0)	истинно (1)
истинно (1)	истинно (1)	истинно (1)

⇒ в очень сложных условиях иногда используется операция **"НЕ"** — отрицание условия (или обратное условие)

! условие

Например, следующие два условия равносильны

A > B ! (A <= B)

- Порядок выполнения (*приоритет*) логических операций и отношений:

- ⇒ операции в скобках, затем
- ⇒ операция "НЕ", затем
- ⇒ логические отношения $>$, $<$, $>=$, $<=$, $==$, $!=$, затем
- ⇒ операция "И", затем
- ⇒ операций "ИЛИ"

- Для изменения порядка действий используются круглые скобки.

Досрочный выход из цикла

Иногда надо выйти из цикла и перейти к следующему оператору, не дожидаясь окончания очередного шага цикла. Для этого используют специальный оператор `break`. Можно также сказать компьютеру, что надо завершить текущий шаг цикла и сразу перейти к новому шагу (не выходя из цикла) — для этого применяют оператор `continue`.

Задача. Написать программу, которая вычисляет частное и остаток от деления двух введенных целых чисел. Программа должна работать в цикле, то есть запрашивать значения делимого и делителя, выводить результат, снова запрашивать данные и т.д. Если оба числа равны нулю, надо выйти из цикла и завершить работу программы. Предусмотреть сообщение об ошибке в том случае, если второе число равно нулю, а первое — нет.

Особенность этой задачи состоит в том, что при входе в цикл мы не можем определить, надо ли будет выполнить до конца очередной шаг. Необходимая информация поступает лишь при вводе данных с клавиатуры. Поэтому здесь используется бесконечный цикл `while(1){...}` (единица считается истинным условием). Выйти из такого цикла можно только с помощью специального оператора `break`.

В то же время, если второе число равно нулю, то оставшуюся часть цикла не надо выполнять. Для этого служит оператор `continue`.

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int A, B;
    while ( 1 )
    {
        printf ( "\nВведите делимое и делитель:" );
        scanf ( "%d%d", &A, &B );

        if ( A == 0 && B == 0 ) break;
        if ( B == 0 ) {
            printf ( "Деление на ноль" );
            continue;
        }

        printf ( "Частное %d остаток %d", A/B, A%B )
    }
    getch();
}
```

бесконечный цикл

выход из цикла

досрочный переход
в начало цикла

Что новенького?

- Если только внутри цикла можно определить, надо ли делать вычисления в теле цикла и надо ли продолжать цикл (например, при вводе исходных данных), часто используют бесконечный цикл, внутри которого стоит оператор выхода `break`:

```
while ( 1 ) {
...
if ( надо выйти ) break;
...
}
```

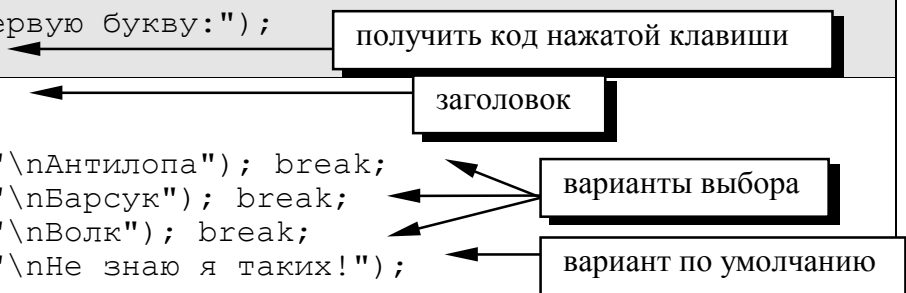
- С помощью оператора `break` можно досрочно выходить из циклов `for`, `while`, `do – while`.
- Чтобы досрочно завершить текущий шаг цикла и сразу перейти к следующему шагу, используют оператор `continue`.

Переключатель `switch` (множественный выбор)

Если надо выбрать один из нескольких вариантов в зависимости от значения некоторой целой или символьной переменной, можно использовать несколько вложенных операторов `if`, но значительно удобнее использовать специальный оператор ***switch***.

Задача. Составить программу, которая вводит с клавиатуры русскую букву и выводит на экран название животного на эту букву.

```
#include <stdio.h>
#include <conio.h>
void main()
{
char c;
printf("\nВведите первую букву:");
c = getch();
switch ( c )
{
case 'a': printf("\nАнтилопа"); break;
case 'б': printf("\nБарсук"); break;
case 'в': printf("\nВолк"); break;
default: printf("\nНе знаю я таких!");
}
getch();
}
```



Что новенького?

- Функция `getch` возвращает в качестве результата код нажатой клавиши в таблице кодов. Код клавиши - это целое число от 0 до 255.
- Оператор множественного выбора `switch` состоит из заголовка и тела оператора, заключенного в фигурные скобки.
- В заголовке после ключевого слова `switch` в круглых скобках записано имя переменной (целой или символьной), в зависимости от значения которой делается выбор между несколькими вариантами.
- Каждому варианту соответствует метка `case`, после которой стоит одно из возможных значений этой переменной и двоеточие; если значение переменной совпадает с одной из меток, то программа переходит на эту метку и выполняет все последующие операторы.
- Оператор `break` служит для выхода из тела оператора `switch`.

- Если убрать все операторы `break`, то, например, при нажатии на букву `'a'` будет напечатано
Антилопа
Барсук
Волк
- Если значение переменной не совпадает ни с одной из меток, программа переходит на метку `default` (по умолчанию, то есть если ничего другого не указано).
- Можно ставить две метки на один оператор, например, чтобы программа реагировала как на большие, так и на маленькие буквы, надо в теле оператора `switch` написать так:

```
case 'a':  
case 'A':  
    printf("\nАнтилопа"); break;  
case 'б':  
case 'Б':  
    printf("\nБарсук"); break;
```

и так далее.

5. Методы отладки программ

Отладочные средства *Borland C 3.1*

Пошаговое выполнение

Обычно программа запускается на выполнение клавишами **Ctrl-F9** и выполняется безостановочно. Если вы заметили, что она работает не так, как вы хотели, ее надо *отлаживать*, то есть искать и исправлять в ней ошибки.

Лучший способ отладки — это выполнить программы по строкам, останавливаясь после выполнения каждой строки и проверяя значения переменных в памяти. Для этой цели служат специальные программы — *отладчики*.

Отладчики бывают автономные и встроенные. С системой программирования *Borland C* поставляются автономные отладчики *Turbo Debugger* (файлы *td.exe*, *td286.exe* и *td386.exe*). Однако для отладки простых программ удобнее использовать встроенный отладчик оболочки *Borland C*, в которой вы набираете и запускаете программу. Для того, чтобы с помощью встроенного отладчика выполнить программу по строкам, служит комбинация клавиш **F8**.

Полоска голубого цвета обозначает строку, которая будет выполняться следующей (но еще не выполнялась). Если нажать на **F8**, выполняются все операторы в этой строке и выделение переходит на следующую строку. Для запуска программы без остановки надо нажать **Ctrl-F9**, для остановки программы — **Ctrl-F2**.

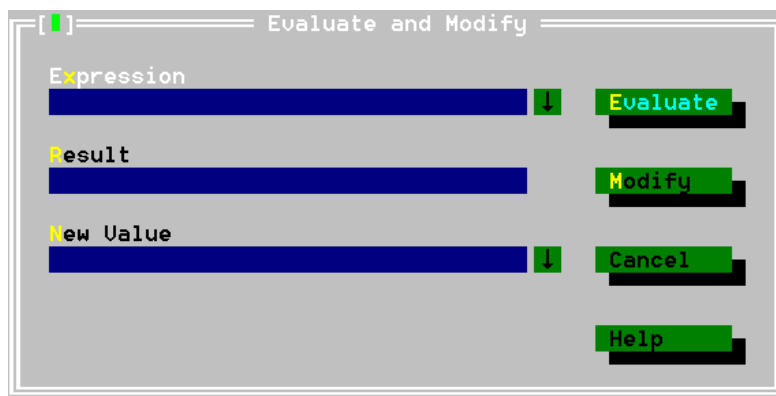
Однако в этом режиме встроенный отладчик не позволяет входить внутрь вызываемых процедур (по шагам выполняется только основная программа). Для входа в процедуру или функцию используют комбинацию **F7** (если в этой строке нет вызова процедур и функций, она равносильна **F8**).

Просмотр и изменение значений переменных

Пошаговое выполнение программы позволяет лишь посмотреть, какие операторы программы выполняются, сколько раз и в какой последовательности. Однако чаще всего этого оказывается недостаточно для обнаружения ошибки.

Очень мощным средством отладки является просмотр значений переменных во время выполнения программы. Для этого надо нажать **Ctrl-F4**, после чего появится такое окно.

Если в этом окне в поле **Expression** ввести имя переменной или массива и нажать клавишу **Enter**, то соответствующее значение появится в поле **Result**. Более того, если вы захотите изменить значение этой переменной (и посмотреть, как будет выполняться программа дальше, если бы значение этой переменной было бы другое), надо перейти в поле **New Value**, ввести там новое значение и нажать **Enter**.



Окно просмотра переменных

В интегрированной среде *Borland C* существует специальное окно, в котором вы можете постоянно наблюдать значения переменных во время выполнения программы. Для добавления

новой переменной в это окно просмотра надо нажать **Ctrl-F7**, ввести имя переменной и нажать **Enter**. Чтобы перейти в окно просмотра, надо выбрать в меню пункты **Window-Watch** (или нажать клавиши **Alt-W + W**).

Точки останова (Breakpoints)

Встроенный отладчик позволяет также остановить программу в любом месте. Для этого надо поставить курсор в нужную строчку и нажать **Ctrl-F8**. При этом строка выделяется красным цветом (для снятия точки останова надо еще раз нажать **Ctrl-F8**). Программа, запущенная по **Ctrl-F9**, выполняется без остановки, пока не встретится точка останова. В этом случае вы снова оказываетесь в окне редактора и можете просмотреть значения переменных и продолжить программу в пошаговом режиме.

Существуют еще дополнительные возможности — задать останов в некоторой строке при определенном условии или после прохождения через эту точку определенное количество раз. Это выполняется с помощью меню **Debug—Breakpoints-Edit**.

Отладочные комбинации клавиш

Alt-F9	компиляция текущего модуля (проверка ошибок)
F9	компиляция всех модулей проекта и компоновка выполняемой программы (<i>exe</i> -файла)
Ctrl-F9	запуск программы (если надо, то она компилируется и компоуется)
F8	выполнить один шаг и остановиться
F7	войти в процедуру
Alt-F5	просмотреть экран программы
Ctrl-F2	остановить пошаговое выполнение
F4	выполнить до строки, в которой стоит курсор
Ctrl-F4	просмотр и изменение значений переменных
Ctrl-F7	добавить переменную в окно просмотра

Практические приемы

Отключение частей кода

Довольно часто при внесении каких-то изменений программа перестает работать. Основная задача в этом случае — определить, в каком именно месте возникает ошибка (иначе говоря, с какого места программа перестает работать верно). При этом хорошие результаты дает *метод отключения блоков программы*: начиная с конца программы, новые блоки заключаются в знаки комментария (*/* и */*), то есть компьютер их не выполняет. Таким образом, добиваются, чтобы программа работала правильно, и определяют строку, в которой происходит ошибка.



Ручная прокрутка программы

Если другие способы не помогают, приходится делать *ручную прокрутку программы*, то есть выполнять программу вручную вместо компьютера, записывая результаты на лист бумаги.

Обычно составляют таблицу, в которую записывают изменения всех переменных (неизвестное значение переменной обозначают знаком вопроса). Рассмотрим (ошибочную) программу, которая вводит натуральное число и определяет, простое оно или нет. Мы выяснили, что она дает неверный результат при $N=5$ (печатает, что 5 – якобы составное число). Построим таблицу изменения значений переменных для этого случая.

```
#include <stdio.h>
void main()
{
    int    N, i, count = 0;
    printf("Введите число ");
    scanf("%d", &N);
    for ( i = 2; i <= N; i ++ )
        if ( N % i == 0 )
            count ++;
    if ( count == 0 )
        printf("Число простое");
    else printf("Число составное");
}
```

N	i	count
2	?	0
	2	
	3	
	4	
	5	1

Выполняя вручную все действия, выясняем, что программа проверяет делимость числа N на само себя, то есть, счетчик делителей `count` всегда будет не равен нулю. Теперь, определив причину ошибки, легко ее исправить. Для этого достаточно заменить условие в цикле на $i < N$.



Проверка крайних значений

Очень важно проверить работу программы (и функций) на крайних значениях возможного диапазона исходных данных. Например, для программы, определяющей простоту натуральных чисел, надо проверить, будет ли она работать для чисел **1** и **2**. Аналогично, если мы отлаживаем программу, которая ищет заданное слово в строке, надо проверить ее для тех случаев, когда искомое слово стоит первым или последним в строке.

6. Работа в графическом режиме

Общая структура программы

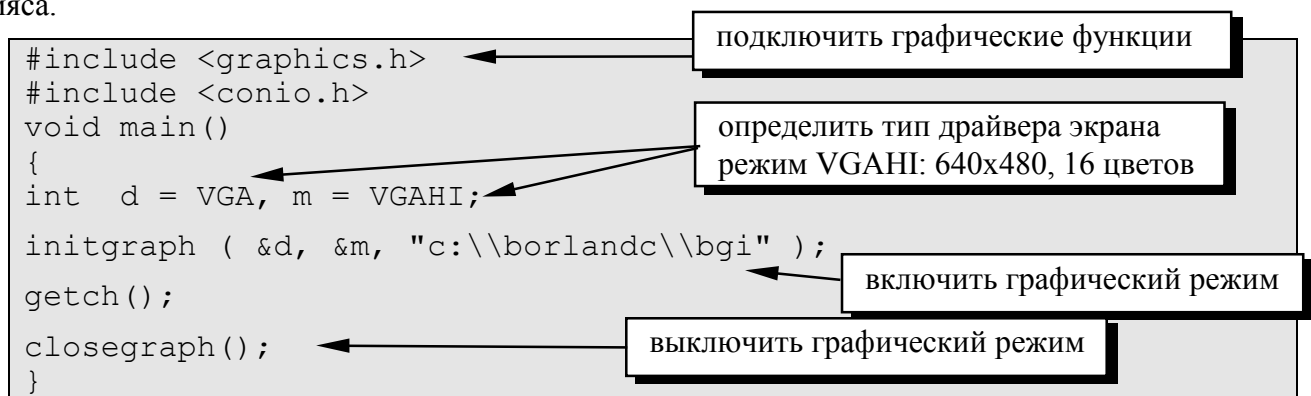
Особенность современных компьютеров заключается в том, что они могут работать в двух режимах — текстовом и графическом, поэтому для рисования на экране линий, прямоугольников и т.п. надо включать монитор в графический режим. При этом текст также можно выводить на экран, но с помощью других функций. Таким образом, графическая программа на Си имеет структуру сэндвича:



При переключении режима (из текстового в графический или наоборот) все содержимое экрана стирается. Поэтому перед выходом из графического режима надо делать паузу до нажатия клавиши — иначе не увидеть результат работы программы.

Простейшая графическая программа

Сейчас мы напишем простейшую графическую программу. Она не делает ничего полезного, просто включает монитор в графический режим, делает паузу до нажатия клавиши и выключает графический режим. Поэтому программа эта так же неполноценна, как сэндвич без мяса.



Что новенького?

- Для использования графических функций надо подключить файл заголовков **graphics.h** в начале программы.
- Для включения монитора в графический режим надо объявить две целых переменные:
 - ⇒ Одна из них (в примере — **d**) обозначает тип драйвера монитора (**драйвер** — это специальный набор процедур, который выполняет графические операции). Эта переменная может принимать значения **CGA**, **EGA**, **VGA** и **DETECT** (определить тип монитора автоматически). Все современные мониторы поддерживают стандарт **VGA**, поэтому мы будем использовать его.

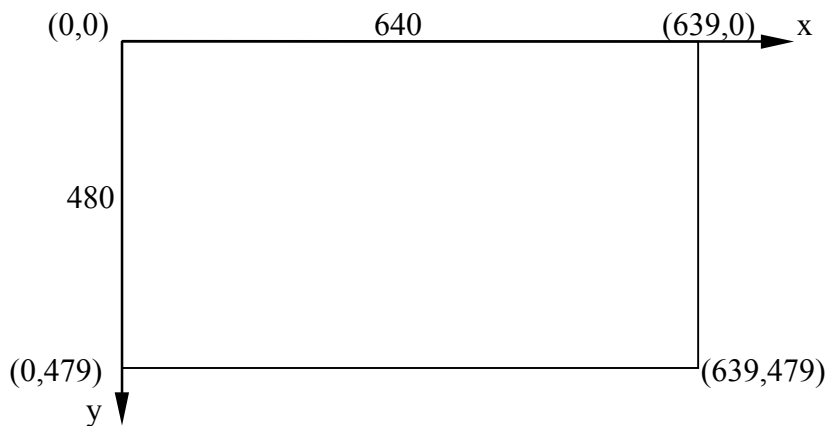
⇒ Вторая переменная (в примере — *m*) обозначает тип графического режима, например, *VGAHI* обозначает режим 640 на 480 точек, 16 цветов.

- Можно попросить, чтобы графический режим был выбран автоматически. Для этого в переменную *d* надо записать значение **DETECT** (все большие буквы), а значение переменной *m* не играет роли.
- Функция `initgraph()` включает графический режим. Она имеет 3 параметра: адреса описанных двух выше переменных и строка, в которой содержится путь к файлу-драйверу заданного графического режима (или пустая строка, если этот файл находится в текущем каталоге).
- В строках языка Си обратный слэш '`\`' — это специальный символ (вспомните '`\n`' — переход на новую строку), поэтому он дублируется, если надо включить в строку сам этот символ.
- Драйверы графического режима имеют расширение **.bgi* и находятся в подкаталоге *BGI* того каталога, в который установлен компилятор Си. Для мониторов *EGA* и *VGA* используется драйвер *egavga.bgi*.
- Графический режим выключается с помощью функции `closegraph`. При этом экран очищается и монитор переключается в текстовый режим.



Графические функции

Пришло время нарисовать что-то на экране. Для этого надо представлять, как определять координаты на графическом экране.



Что новенького?

- В режиме *VGAHI* экран имеет размер 640 на 480 точек (*пикселей*).
- Начало координат, точка (0,0), находится в левом верхнем углу экрана.
- Ось *x* направлена влево, ось *y* — вниз (в отличие от общепринятой математической системы координат).
- В режиме *VGAHI* можно использовать 16 цветов, каждый из них имеет цифровое и символьное обозначения (лучше в программах использовать символьное обозначение, которое более понятно):

0	BLACK	черный	8	DARKGRAY	темно-серый
1	BLUE	синий	9	LIGHTBLUE	светло-синий
2	GREEN	зеленый	10	LIGHTGREEN	светло-зеленый
3	CYAN	морской волны	11	LIGHTCYAN	светлый морской волны
4	RED	красный	12	LIGHTRED	светло-красный
5	MAGENTA	фиолетовый	13	LIGHTMAGENTA	светло-фиолетовый
6	BROWN	коричневый	14	YELLOW	желтый
7	LIGHTGRAY	светло-серый	15	WHITE	белый

Для рисования используются стандартные функции

<code>putpixel (x, y, color);</code>	установить точке (x,y) цвет
<code>n = getpixel (x, y);</code>	получить цвет точки (x,y) и записать его в целую переменную n
<code>setcolor(color);</code>	установить цвет линий color
<code>line (x1, y1, x2, y2);</code>	отрезок (x₁,y₁) - (x₂,y₂)
<code>rectangle (x1, y1, x2, y2);</code>	прямоугольник: левый верхний угол (x₁,y₁) , правый нижний — (x₂,y₂)
<code>circle (x, y, R);</code>	окружность с центром радиуса R
<code>setfillstyle (SOLID_FILL, color);</code>	установить режим сплошной заливки цветом color (влияет на работу функций <code>bar</code> , <code>floodfill</code> и некоторых других)
<code>bar (x1, y1, x2, y2);</code>	залитый прямоугольник
<code>floodfill (x, y, color);</code>	залить область, которая содержит точку (x,y) и ограничена замкнутой линией цвета color
<code>outtextxy (x, y, "Привет!");</code>	вывести текст, левый верхний угол которого будет находиться в точке (x,y)



Пример программы

Напишем программу, которая использует стандартные функции для рисования на белом фоне красного прямоугольника с синей границей и синими диагоналями, и желтого круга с фиолетовой границей в центре экрана. Внутри круга вывести текст "Привет" зеленого цвета.

```

#include <graphics.h>
#include <conio.h>

void main()
{
    int d = DETECT, m;
    initgraph ( &d, &m, "C:\\\\BORLANDC\\\\BGI" );
    setfillstyle (SOLID_FILL, WHITE);
    bar ( 0, 0, 639, 479 );
    setfillstyle ( SOLID_FILL, RED);
    bar ( 220, 160, 420, 320 );
    setcolor ( BLUE );
    rectangle ( 220, 160, 420, 320 );
    line ( 220, 160, 420, 320 );
    line ( 220, 320, 420, 160);
    setcolor ( MAGENTA );
    circle ( 320, 240, 50);
    setfillstyle (SOLID_FILL, YELLOW);
    floodfill ( 320, 240, MAGENTA );
    setcolor ( GREEN );
    outtextxy ( 295, 235, "Привет!" );
    getch();
    closegraph();
}

```

автоматически выбрать режим

закрашенный прямоугольник

контур прямоугольника

диагонали

контур окружности

заливка окружности

надпись зеленого цвета



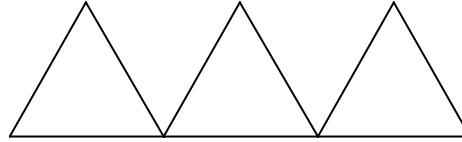
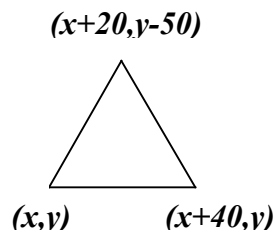
Что новенького?

- Чтобы сделать нужный фон, надо залить весь экран выбранным цветом с помощью функции `bar`.
- Цвет линий надо устанавливать с помощью `setcolor` до рисования линий, а цвет заливки — с помощью `setfillstyle` до вызова функции `bar` или `floodfill`.
- Установленные цвета линий и заливки остаются рабочими до тех пор, пока не будут переустановлены с помощью `setcolor` или `setfillstyle`.

7. Процедуры

Пример задачи с процедурой

Часто в программах бывает легко выделить одинаковые элементы (например, одинаковые фигуры в рисунке). Составим программу, которая рисует на экране три одинаковых треугольника.



Единственное, чем отличаются эти треугольники — это место на экране. Его удобно задать координатами (x,y) одного из его углов. Пусть (x,y) — координаты левого нижнего угла треугольника, его высота 50 пикселей и основание 40 пикселей. Координаты вершин такого треугольника показаны на рисунке (с учетом того, что ось y направлена вниз). Таким образом, надо нарисовать три линии заданного цвета.

Конечно, можно три раза подряд написать почти одинаковые строчки и получить нужный результат. Однако лучше всего научить компьютер рисовать такие треугольники — ввести новую команду (к сожалению, ее название надо писать латинскими буквами) и расшифровать ее, а затем вызывать эту команду, указывая координаты левого нижнего угла для каждого нового треугольника. Такие новые команды, введенные программистом, называют *процедурами*.

Процедура - это вспомогательная программа (*подпрограмма*), предназначенная для выполнения каких-либо действий, которые встречаются в нескольких местах программы.

Решение нашей задачи выглядит так:

```
#include <graphics.h>
#include <conio.h>

void triangle ( int x, int y ); ← объявление процедуры

void main()
{
    int    grDriver = VGA, grMode = VGAHI;
    initgraph ( &grDriver, &grMode, "C:\\BORLANDC\\BGI" );

    triangle ( 100, 100 );
    triangle ( 140, 100 );
    triangle ( 140, 100 ); ← вызовы процедуры

    getch();
    closegraph();
}

void triangle ( int x, int y )
{
    line ( x, y, x + 20, y - 50 );
    line ( x, y, x + 40, y );
    line ( x + 40, y, x + 20, y - 50 );
} ← тело процедуры
```



Что новенького?

- Процедура оформляется так же, как основная программа: заголовок и тело процедуры в фигурных скобках.
- Перед именем процедуры ставится слово `void`. Это означает, что она не возвращает результат, а только выполняет какие-то действия.
- После имени в скобках через запятую перечисляются **параметры** процедуры — те величины, от которых зависит ее работа.
- Для каждого параметра отдельно указывается его тип (`int`, `float`, `char`).
- Имена параметров можно выбирать любые, допустимые в языке Си.
- Параметры, перечисленные в заголовке процедуры, называются **формальными** — это значит, что они доступны только внутри процедуры при ее вызове.
- Желательно выбирать осмысленные имена параметров процедуры — это позволяет легче разобраться в программе потом, когда уже все забыто.
- При вызове процедуры надо указать ее имя и в скобках **фактические** параметры, которые подставляются в процедуру вместо формальных параметров.
- Фактические параметры — это числа или любые арифметические выражения (в этом случае сначала рассчитывается их значение).
- Первый фактический параметр подставляется в процедуру вместо первого формального параметра, и т.д.
- Все процедуры необходимо объявить до основной программы; это делается для того, чтобы к моменту вызова процедуры транслятор знал, что есть такая процедура, а также сколько она имеет параметров и каких. Это позволяет находить ошибки на трансляции, например такие:

`triangle (100);` **Too few parameters** (слишком мало параметров)

`triangle (100, 100, 5);` **Extra parameter** (лишний параметр)

- При объявлении процедуры после заголовка ставится точка с запятой, а в том месте, где записано тело процедуры — не ставится.
- Часто процедуры вызываются только один раз — в этом случае их задача — разбить большую основную программу (или процедуру) на несколько самостоятельных частей, поскольку рекомендуется, чтобы каждая процедура была длиной не более 50 строк (2 экрана по 25 строк), иначе очень сложно в ней разобраться.
- Для досрочного выхода из процедуры используется оператор `return`, при его выполнении работа процедуры заканчивается.
- Если оператор `return` стоит в основной программе, программа заканчивает работу.
- В процедуре можно использовать несколько операторов `return`: при выполнении любого из них работа процедуры заканчивается.



Улучшение процедуры

Можно попробовать сделать процедуру более универсальной: менять цвет границы, залить треугольник выбранным цветом, менять его высоту и ширину основания. Это означает, что надо ввести в процедуру дополнительные параметры. Их количество не должно быть очень

большим (более 8-10), потому что в этом случае можно легко перепутать их порядок. В нашей улучшенной процедуре будет 6 параметров:

x, y координаты левого нижнего угла
a, h длина основания и высота треугольника
color цвет линий
fillColor цвет заливки внутренней части

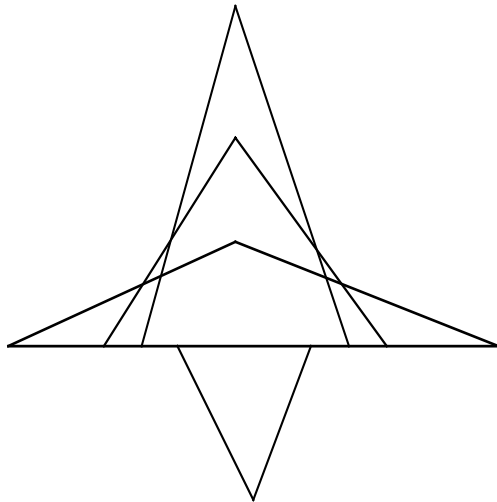
```
void triangle ( int x, int y, int a, int h, int color,
               int fillColor )
{
    setcolor ( color );
    line ( x, y, x + a / 2, y - h );
    line ( x, y, x + a, y );
    line ( x + 40, y, x + a / 2, y - h );
    setfillstyle ( SOLID_FILL, fillColor );
    floodfill ( x + a / 2, y - h / 2, color);
}
```

← изменить цвет линии

← изменить цвет заливки

← заливка

Это позволит нам рисовать значительно более сложные фигуры и раскрашивать их, например, так:



```
triangle ( 100, 100, 40, 60,
          BLUE, GREEN);
triangle ( 90, 100, 60, 40,
          LIGHTBLUE, CYAN);
triangle ( 70, 100, 100, 20,
          LIGHTCYAN, RED);
triangle ( 110, 100, 20, -30,
          YELLOW, WHITE);
```

Из этого примера видно, как сделать, чтобы развернуть треугольник не вверх, а вниз: достаточно просто поставить отрицательную высоту при вызове процедуры.

8. Функции



Отличие функций от процедур

Функции, также как и процедуры, предназначены для выполнения одинаковых операций в разных частях программы. Они имеют одно существенное отличие: задача процедуры вычислить и вернуть в вызывающую программу *значение-результат* (в простейшем случае это целое, вещественное или символьное значение).

Функция - это вспомогательная программа, предназначенная для вычисления некоторой величины. Она также может выполнять какие-то полезные действия.

Покажем использование функции на примере. Решим задачу, которую мы уже решали раньше.

Задача. Написать программу, которая вводит целое число и определяет сумму его цифр. Использовать функцию, вычисляющую сумму цифр числа.

Вспомним, что для того чтобы найти последнюю цифру числа, надо взять остаток от его деления на 10. Затем делим число на 10, отбрасывая его последнюю цифру, и т.д. Сложив все эти остатки-цифры, мы получим сумму цифр числа.

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

int SumDigits ( int N )
{
    int d, sum = 0;
    while ( N != 0 )
    {
        d = N % 10;
        sum = sum + d;
        N = N / 10;
    }
    return sum;
}

void main()
{
    int N, s;
    printf ( "\nВведите целое число " );
    scanf ( "%d", &N );
    s = SumDigits ( N );
    printf ( "Сумма цифр числа %d равна %d\n", N, s );
    getch();
}
```

тело функции

вызов функции



Что новенького?

- Функция оформляется так же, как процедура: заголовок и тело функции в фигурных скобках.
- Перед именем процедуры ставится *тип результата* (int, float, char, и т.д.) — это означает, что она возвращает значение указанного типа.
- После имени в скобках через запятую перечисляются *параметры* функции — те величины, от которых зависит ее работа.
- Для каждого параметра отдельно указывается его тип (int, float, char).

- Имена параметров можно выбирать любые, допустимые в языке Си.
- Параметры, перечисленные в заголовке функции, называются **формальными** — это значит, что они доступны только внутри функции при ее вызове.
- Желательно выбирать осмысленные имена параметров — это позволяет легче разобраться в программе потом.
- При вызове функции надо указать ее имя и в скобках **фактические** параметры, которые подставляются в функции вместо формальных параметров.
- Фактические параметры — это числа или любые арифметические выражения (в этом случае сначала рассчитывается их значение).
- Первый фактический параметр подставляется в функции вместо первого формального параметра, и т.д.
- Для того, чтобы определить значение функции, используется оператор `return`, после которого через пробел записывается возвращаемое значение — число или арифметическое выражение. Примеры:

```
return 34
return s;
return a + 4*b - 5;
```

При выполнении оператора `return` работа процедуры заканчивается.

- В функции можно использовать несколько операторов `return`.
- Если функции находятся ниже основной программы, их необходимо объявить *до* основной программы. Для объявления функции надо написать ее заголовок с точкой с запятой в конце.
- При объявлении функции после заголовка ставится точка с запятой, а в том месте, где записано тело функции — не ставится.



Логические функции

Очень часто надо составить функцию, которая просто решает какой-то вопрос и отвечает на вопрос "Да" или "Нет". Такие функции называются *логическими*. Вспомним, что в Си ноль означает ложное условие, а единица — истинное.

Логическая функция - это функция, возвращающая **1** (если ответ "Да") или **0** (если ответ "Нет").

Логические функции используются в основном в двух случаях:

- Если надо проанализировать ситуацию и ответить на вопрос, от которого зависят дальнейшие действия программы.
- Если надо выполнить какие-то сложные операции и определить, была ли при этом какая-то ошибка.



Простое число или нет?

Задача. Ввести число *N* и определить, простое оно или нет. Использовать функцию, которая отвечает на этот вопрос.

Теперь расположим тело функции ниже основной программы. Чтобы транслятор знал об этой функции во время обработки основной программы, надо объявить ее заранее.

```

#include <stdio.h>
#include <conio.h>
int Simple ( int N );
//--- основная программа ---
void main()
{
    int    N;
    printf ( "\nВведите целое число ");
    scanf ( "%d", &N );
    if ( Simple (N) )
        printf ( "Число %d - простое\n", N );
    else printf ( "Число %d - составное\n", N );
    getch();
}
//--- функция ---
int Simple ( int N )
{
    for ( int i = 2; i*i <= N; i ++ )
        if ( N % i == 0 ) return 0;
    return 1;
}

```

объявление функции

вызов функции

нашли делитель — не простое

нет ни одного делителя



Была ли ошибка?

Задача. Включить монитор в графический режим и в случае ошибки выдать сообщение и завершить работу.

Идея основана на том, что существует функция `graphresult`, которая возвращает *код ошибки* — если она возвращает 0, то ошибки не было, а если не нуль, то включить графический режим не удалось (с помощью функции `grapherrormsg` можно получить текстовое описание ошибки на английском языке).

```

#include <graphics.h>
#include <conio.h>
void main()
{
    int    grDriver = VGA, grMode = VGAHI, errorCode;
    initgraph ( &grDriver, &grMode, "C:\\BORLANDC\\BGI" );
    errorCode = graphresult();
    if ( errorCode ) {
        printf ( "Ошибка: %s", grapherrormsg (errorCode) );
        return;
    }
    getch();
    closegraph();
}

```

получить код ошибки

вывести сообщение об ошибке

выйти из программы



Функции, возвращающие два значения

По определению функция может вернуть только одно значение-результат. Если надо вернуть два и больше результатов, приходится использовать специальный прием — *передачу параметров по ссылке*.

Задача. Написать функцию, которая определяет максимальное и минимальное из двух целых чисел.

В следующей программе используется достаточно хитрый прием: мы сделаем так, чтобы функция изменяла значение переменной, которая принадлежит основной программе. Один результат (минимальное из двух чисел) функция вернет как обычно, а второй — за счет изменения переменной, которая передана из основной программы.

#include <stdio.h> #include <conio.h>	параметр — результат
int MinMax (int a, int b, int &Max) { if (a > b) { Max = a; return b; } else { Max = b; return a; } }	
void main() { int N, M, min, max; printf ("\nВведите 2 целых числа "); scanf ("%d%d", &N, &M);	вызов функции
min = MinMax (N, M, max); printf ("Наименьшее из них %d, наибольшее — %d\n", min, max); getch(); }	

Обычно при передаче параметра в процедуру или функцию в памяти создается копия переменной и процедура работает с этой копией. Это значит, что все изменения, сделанные в процедуре с переменной-параметром, не отражаются на значении этой переменной в вызывающей программе.

Если перед именем параметра в заголовке функции поставить знак **&** (вспомним, что он также используется для определения адреса переменной), то процедура работает прямо с переменной из вызывающей программы, а не с ее копией. Поэтому в нашем примере процедура изменит значение переменной **max** из основной программы и запишет туда максимальное из двух чисел.



Что новенького?

- Если надо, чтобы функция вернула два и более результатов, поступают следующим образом:
⇒ один результат передается как обычно с помощью оператора **return**
⇒ остальные возвращаемые значения передаются через изменяемые параметры
- Обычные параметры не могут изменяться процедурой, потому что она в самом деле работает с *копиями* параметров (например, если менять значения **a** и **b** в процедуре **MinMax**, соответствующие им переменные **N** и **M** в основной программе не изменятся).
- Любая процедура и функция может возвращать значения через изменяемые параметры.
- Изменяемые параметры (или параметры, передаваемые по ссылке) объявляются в заголовке процедуры специальным образом: перед их именем ставится знак **&** — в данном случае он означает ссылку, то есть процедура может менять значение параметра (в данном случае функция меняет значение переменной **max** в основной программе).
- При вызове таких функций и процедур вместо фактических изменяемых параметров надо подставлять только имя переменной (не число и не арифметическое выражение — в этих случаях транслятор выдает предупреждение и формирует в памяти временную переменную).

9. Структура программ



Составные части программы

В составе программы можно выделить несколько частей:

- Подключение *заголовочных файлов* — это строки, которые начинаются с `#include`
- Объявление *констант* (строки `#define` или объявления `const`)

```
#define N 20
```

 или так:

```
const N = 20;
```
- *Глобальные переменные* — это переменные, объявленные вне основной программы и процедур. К таким переменным могут обращаться все процедуры и функции данной программы (их не надо еще раз объявлять в этих процедурах).
- *Объявление функций и процедур* — обычно ставятся выше основной программы. По требованиям языка Си в тот момент, когда транслятор находит вызов процедуры, она должна быть объявлена и известны типы всех ее параметров.
- *Основная программа* — она может располагаться как до всех процедур, так и после них. Не рекомендуется вставлять ее в середину, между процедурами, так как при этом ее сложнее найти.



Глобальные и локальные переменные

Глобальные переменные доступны из любой процедуры или функции. Поэтому их надо объявлять вне всех процедур. Остальные переменные, объявленные в процедурах и функциях, называются *локальными* (местными), поскольку они известны только той процедуре, где они объявлены. Следующий пример показывает различие между локальными и глобальными переменными.

#include <stdio.h>	
int var = 0;	← объявить глобальную переменную
void ProcNoChange ()	
{	
int var;	← меняется только локальная переменная
var = 3;	
}	
void ProcChange1 ()	
{	
var = 5;	← меняется только глобальная переменная
}	
void ProcChange2 ()	
{	
int var;	← меняется локальная переменная
var = 4;	
::var = ::var * 2 + var;	← меняется глобальная переменная
}	
void main()	
{	
ProcChange1();	// var = 5;
ProcChange2();	// var = 5*2 + 4 = 14;
ProcNoChange();	// var не меняется
printf ("%d", var);	// печать глобальной переменной
}	

Что новенького?

- Глобальные переменные не надо заново объявлять в процедурах.
- Если в процедуре объявлена локальная переменная с таким же именем, как и глобальная переменная, то используется *локальная* переменная.
- Если имена глобальной и локальной переменных совпадают, то для обращения к глобальной переменной в процедуре перед ее именем ставится два двоеточия:

```
::var = ::var * 2 + var;
```

↑
глобальная переменная

↑
локальная переменная

Однако специалисты рекомендуют использовать очень мало глобальных переменных или не использовать их вообще, потому что глобальные переменные

- затрудняют анализ и отладку программы
- повышают вероятность серьезных ошибок — можно не заметить, что какая-то процедура изменила глобальную переменную
- увеличивают размер программы, так как заносятся в блок данных, а не создаются в процессе выполнения программы

Поэтому глобальные переменные используют в крайних случаях:

- для хранения глобальных системных настроек (цвета экрана и т.п.)
- если переменную используют три и более процедур и по каким-то причинам неудобно передавать эти данные в процедуру как параметр

Везде, где можно, надо передавать данные в процедуры и функции через их параметры. Если же надо, чтобы процедура меняла значения переменных, надо передавать параметр по ссылке.

Оформление текста программы

Зачем оформлять программы?

Зачем же красиво и правильно оформлять тексты программ? На этот вопрос вы сможете ответить сами, сравнив две абсолютно одинаковые (с точки зрения транслятора) программы:

```
#include <stdio.h>
void main()
{
    float      x, y;
    printf ("\nВведите 2 числа ");
    scanf ("%d%d", &x, &y);
    printf ("Их сумма %d ", x+y);
}
```

```
#include <stdio.h> void
main() { float x, y;
printf ( "\nВведите 2
числа " ); scanf (
"%d%d", &x, &y ); printf
( "Их сумма %d ", x+y );}
```

То, что в них отличается и называется грамотным оформлением (очевидно, что оно присутствует в первой программе).

Оформление текста программы необходимо для того, чтобы

- отлаживать программу (искать и исправлять ошибки в ней)
- разбираться в алгоритме работы программы



Оформление процедур и функций

При оформлении функций и процедур рекомендуется придерживаться следующих правил:

- Имена функций и процедур должны быть информативными, то есть нести информацию о том, что делает эта функция. К сожалению, транслятор не понимает русские имена, поэтому приходится писать по-английски. Если вам сложно писать имена на английском языке, можно писать русские слова английскими буквами. Например, процедуру, рисующую квадрат, можно объявить так:

```
void Square ( int x, int y, int a );
```

или так

```
void Kvadrat ( int x, int y, int a );
```

- Перед заголовком процедуры надо вставлять несколько строк с комментарием (здесь можно писать по-русски). В комментарий записывается самая важная информация: что означают параметры функции, что она делает и какое значение она возвращает, ее особенности.
- Одинаковые операции в разных частях программы оформляются в виде процедур и функций
- Не рекомендуется делать функции длиной более 25-30 строк, так как при этом они не помещаются на один экран монитора (25 строк), становятся сложными и запутанными. Если функция получается длинной, ее надо разбить на более мелкие процедуры и функции.
- Чтобы отделить одну часть функции от другой используют пустые строки. Крупный смысловой блок функции можно выделять строкой знаков минус в комментариях.

Приведем пример оформления на примере функции, которая рисует на экране закрашенный ромб с заданными параметрами, если весь ромб помещается на экран (при этом результат функции равен 1) или возвращает признак ошибки (число 0).

```
//*****
// ROMB – рисование ромба в заданной позиции
//      (x,y)      – координаты центра ромба
//      a, b – ширина и высота ромба
//      color, colorFill – цвета границы и заливки
//      Возвращает 1, если операция выполнена, и 0 если
//      ромб выходит за пределы экрана
//*****
int Romb ( int x, int y, int a, int b, int color,
          int colorFill )
{
    if ( (x < a) || (x > 640-a) || (y < a) || (y > 480-b) )
        return 0;
//-----
    setcolor ( color );
    line ( x-a, y, x, y-b );      line ( x-a, y, x, y+b );
    line ( x+a, y, x, y-b ); line ( x+a, y, x, y+b );

    setfillstyle ( SOLID_FILL, colorFill );
    floodfill ( x, y, color );

    return 1;
}
```

Diagram illustrating the structure of the `Romb` function with annotations:

- заголовок** (header): Points to the comment block above the function signature.
- обработка ошибки** (error handling): Points to the `if` statement that checks for boundary conditions and returns 0.
- крупный блок** (large block): Points to the drawing logic (lines and fill) separated by a dashed comment line.
- выделение пустой строкой** (separation by empty line): Points to the empty line between the error handling and the drawing logic.



Отступы

Отступы используются для выделения структурных блоков программы (процедур, циклов, условных операторов). Отступы позволяют легко искать лишние и недостающие скобки, понимать логику работы программы и находить в ней ошибки. При расстановке отступов рекомендуется соблюдать следующие правила:

- Величина отступа равна 2 — 3 символа.
- Все тело любой процедуры и функции имеет один отступ.
- Дополнительным отступом выделяются
 - ⇒ тело циклов `for`, `while`, `do-while`
 - ⇒ тело условного оператора `if` и блока `else`
 - ⇒ тело оператора множественного выбора `switch`
- Парные скобки должны находиться на одной вертикали (чтобы найти скобку, парную данной, надо поставить на эту скобку курсор и нажать на клавиши **Ctrl-Q** и затем `{` или `}`.

10. Анимация



Что такое анимация?

Анимация – это оживление изображения на экране (от английского слова *animate* - оживлять). При этом объекты движутся, вращаются, сталкиваются, меняют форму и цвет и т.д.

Чтобы сделать программу с эффектами анимации, надо решить две задачи:

- двигать объект так, чтобы он мигал как можно меньше
- обеспечить управление клавишами или мышкой во время движения

В этом разделе рассматриваются самые простые задачи этого типа. Во всех программах предусмотрен выход по клавише *Escape*.



Движение объекта

Составим программу, которая передвигает по экрану какой-то объект (в нашем случае – равнобедренный треугольник) от левой границы экрана к правой. Программа заканчивает работу, когда объект вышел за границы экрана или была нажата клавиша *Escape*.



Предварительный анализ

Рассмотрим объект, который движется по экрану. Пусть это будет равнобедренный треугольник с основанием 20 пикселей и высотой также 20 пикселей. Он будет двигаться по экрану, поэтому координаты всех вершин будут меняться. Чтобы строить треугольник в любом месте экрана, мы выберем одну из точек в треугольнике в качестве базовой (главной) и обозначим ее координаты за (x, y) . Теперь выразим через них координаты всех вершин (см. рисунок).

Теперь надо придумать способ изобразить движение так, чтобы рисунок не мигал и программа работала одинаково на всех компьютерах независимо от их быстродействия. Для этого применяют такой алгоритм:

1. рисуем фигуру на экране
2. делаем небольшую задержку (обычно 10-20 мс)
3. стираем фигуру
4. меняем ее координаты
5. переходим к началу алгоритма

Эти действия повторяются до тех пор, пока не будет получена команда «закончить движение» (в нашем случае - нажата клавиша *Escape* или объект вышел за правую границу экрана).

Пусть движение треугольника происходит на черном фоне. Тогда самый быстрый и простой способ стереть его – это нарисовать его же, но черным цветом. Поэтому удобно написать процедуру, параметрами которой являются координаты x и y , а также цвет линий *color*. Когда мы используем черный цвет, фигура стирается с экрана.

Все действия, которые входят в алгоритм, надо выполнить много раз, поэтому используем цикл. Кроме того, мы заранее не знаем, сколько раз должен выполняться этот цикл, поэтому применяем цикл *while* (цикл с условием).

Условие окончания цикла – выход фигуры за границы экрана или нажатие на клавишу *Escape*. Мы будем работать в графическом режиме с разрешением 640 на 480 пикселей, где

координата x может меняться от 0 до 639, поэтому условие выхода треугольника за границы экрана имеет вид $x + 20 \geq 640$, а нужное нам условие продолжения цикла:

```
x + 20 < 640
```

Обработка событий клавиатуры

Надо также обеспечить выход по клавише **Escape**. При этом объект должен двигаться и нельзя просто ждать нажатия на клавишу с помощью функции `getch`. В этом случае используют следующий алгоритм:

1. Проверяем, нажата ли какая-нибудь клавиша; это делает функция `kbhit`, которая возвращает результат 1 (ответ «да»), если клавиша нажата, и результат 0 (ответ «нет»), если никакая клавиша не нажата. В программе проверка выполнена с помощью условного оператора

```
if ( kbhit() ) { ... }
```

2. Если клавиша нажата, то

- Определяем код этой клавиши, вызывая функцию `getch`. Код клавиши – это ее номер в таблице символов. Если на символ отводится 1 байт памяти, то всего можно использовать 256 разных символов и их коды изменяются в интервале от 0 до 255.
- Если полученный код равен коду клавиши **Escape** (27), то выходим из цикла

Для того, чтобы управлять программой с помощью клавиш, надо использовать их коды. Вот некоторые из них

Escape	27
Enter	13
пробел	32

Программа

Вообще говоря, в процедуре можно рисовать любую фигуру. В нашем случае рисуем равнобедренный треугольник. Если процедура расположена ниже основной программы, ее надо объявить заранее.

Первые две строки основной программы – включение графического режима. Затем объявляются переменные x , y – координаты фигуры, и dx – величина смещения фигуры за 1 шаг. После этого устанавливаем начальные значения всех переменных.

В цикле `while` условием продолжения будет $x + 20 < 640$, то есть выполнять пока фигура находится в пределах экрана. Нажатие на клавишу **Escape** обрабатывается внутри цикла. Сначала мы определяем, нажата ли какая-нибудь клавиша (с помощью функции `kbhit`), затем определяем ее код (функция `getch`) и, если он равен коду **Escape**, выходим из цикла с помощью оператора `break`.

В основной части цикла рисуем фигуру с помощью процедуры, затем делаем задержку на 20 мс, вызывая функцию `delay` с параметром 20, и стираем фигуру. Для использования функции `delay` надо подключить дополнительную библиотеку функций, добавив в начало программы строку

```
#include <dos.h>
```

После этого изменяем координаты и возвращаемся к началу цикла. Переменная dx обозначает величину смещения за 1 шаг. Если увеличить dx , фигура будет двигаться быстрее, но скачками. Заметим, что если dx станет отрицательным, то фигура будет двигаться влево, потому что ее координата x уменьшается.

```

#include <graphics.h>
#include <conio.h>
#include <dos.h>

void Figure ( int x, int y, int color )
{
    setcolor ( color );
    line ( x, y, x+20, y );
    line ( x, y, x+10, y-20 );
    line ( x+10, y-20, x+20, y );
}

void main()
{
    int d = VGA, m = VGAHI;
    int x, y, dx;
    initgraph ( &d, &m, "c:\\borlandc\\bgi" );
    x = 0; y = 240;
    dx = 1;
    while ( x + 20 < 640 )
    {
        if ( kbhit() )
            if ( getch() == 27 ) break;
        Figure ( x, y, YELLOW );
        delay ( 20 );
        Figure ( x, y, BLACK );
        x += dx;
    }
    closegraph();
}

```

функция рисует

начальные координаты

шаг

если нажата клавиша...

если нажали *Esc*, выход

рисуем фигуру

задержка 20 мс

стираем фигуру

двигаем фигуру

Что новенького?

- Чтобы определить нажата ли какая-нибудь клавиша, используется функция `kbhit`. Она возвращает 1, если клавиша нажата, или 0, если нет.
- Если клавиша уже была нажата, ее код можно получить с помощью функции `getch`.
- Чтобы сделать задержку на заданное время, используется процедура `delay`, которая описана в файле `dos.h`. Параметром этой процедуры является величина задержки в миллисекундах. Если уменьшать задержку, фигура будет двигаться быстрее.

Отскок от поверхности

Теперь сделаем так, чтобы треугольник не уходил за границы экрана, а «отскакивал» от них и начинал движение в другую сторону, и так до тех пор, пока не нажата клавиша *Escape*.

В предыдущей программе была использована переменная ***dx***, которая задает смещение фигуры за 1 шаг. Если ***dx* > 0**, то фигура движется вправо (координата *x* увеличивается), а если ***dx* < 0**, то влево. Таким образом, чтобы сделать отскок от правой стенки, надо в момент «столкновения» (то есть, при ***x* + 20 >= 639**) поменять ***dx*** на некоторое отрицательное значение. Аналогично, во время отскока от левой стенки (то есть, при ***x* <= 0**) значение ***dx*** должно стать положительным.


```

while ( 1 )
{
    if ( kbhit() )
        if ( getch() == 27 ) break;

    Figure ( x, y, YELLOW );
    delay ( 10 );
    Figure ( x, y, BLACK );

    if ( x + 20 >= 639 ) dx = - 1;
    if ( x <= 0 )        dx = 1;

    x += dx;
}

```

бесконечный цикл

меняем направление

Здесь показан только основной цикл, поскольку оставшаяся часть программы не изменяется.

Заметим, что изменился заголовок цикла `while`, теперь вместо условия стоит единица. В языке Си это означает истинное условие, то есть всегда верное. Таким образом, мы получили бесконечный цикл. Чтобы выйти из него, надо нажать клавишу *Esc*. При этом программа выходит на оператор `break` и цикл заканчивается.



Управление клавишами-стрелками

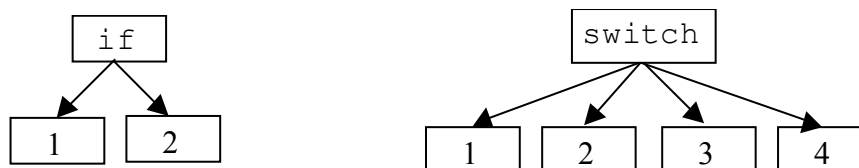


Предварительный анализ

Принцип работы программы очень простой: получив код клавиши, надо сдвинуть объект в соответствующую сторону. Мы введем две переменные ***dx*** и ***dy***, которые будут обозначать величину изменения координат фигуры *x* и *y* за 1 шаг цикла. У нас обрабатываются 4 варианта:

движение влево	<i>dx</i> > 0, <i>dy</i> = 0
движение вправо	<i>dx</i> < 0, <i>dy</i> = 0
движение вверх	<i>dx</i> = 0, <i>dy</i> < 0
движение вниз	<i>dx</i> = 0, <i>dy</i> > 0

Это значит, что надо сделать выбор одного из четырех вариантов в зависимости от кода нажатой клавиши. Для этого можно использовать несколько условных операторов `if`, но существует специальный оператор `switch`, который позволяет легко организовать выбор из нескольких вариантов.



Еще одна проблема связана с тем, что клавиши управления курсором (стрелки) – не совсем обычные клавиши. Они относятся к группе функциональных клавиш, у которых нет кодов в таблице символов *ASCII*. Когда нажата одна из специальных клавиш, система реагирует на нее как на 2 нажатия, причем для первого код символа всегда равен нулю, а для второго мы получим специальный код (так называемый *скан-код*, номер клавиши на клавиатуре). Мы будем использовать упрощенный подход, когда анализируется только этот второй код:

влево	75	вверх	72
вправо	77	вниз	80

У такого приема есть единственный недостаток: объект будет также реагировать на нажатие клавиш с кодами 75, 77, 72 и 80 в таблице символов, то есть на заглавные латинские буквы К, М, Н и Р.

Простейший случай

Составим программу, при выполнении которой фигура будет двигаться только тогда, когда мы нажмем на клавишу-стрелку. В цикле мы сначала рисуем фигуру, ждем нажатия на клавишу и принимаем ее код с помощью функции `getch`. После этого стираем фигуру в том же месте (пока не изменились координаты) и, в зависимости от этого кода, меняем координаты фигуры нужным образом.

```
#include <graphics.h>
#include <conio.h>
void Figure ( int x, int y, int color )
{
    ... // здесь записываем ту же самую функцию, что и раньше
}
void main()
{
    int d = VGA, m = VGAHI;
    int x, y, key;
    initgraph ( &d, &m, "c:\\borlandc\\bgi" );
    x = 320; y = 240;
    while ( 1 )
    {
        Figure ( x, y, YELLOW );
        key = getch();
        if ( key == 27 ) break;
        Figure ( x, y, BLACK );
        switch ( key ) {
            case 75: x --; break;
            case 77: x ++; break;
            case 72: y --; break;
            case 80: y ++;
        }
    }
    closegraph();
}
```

В операторе `switch` значения координат меняются на единицу, хотя можно использовать любой шаг. В конце обработки каждого варианта надо ставить оператор `break`, чтобы не выполнялись строки, стоящие ниже.

Непрерывное движение

Теперь рассмотрим более сложный случай, когда объект продолжает движение в выбранном направлении даже тогда, когда ни одна клавиша не нажата, а при нажатии клавиши-стрелки меняет направление. Здесь надо использовать переменные ***dx*** и ***dy***, которые задают направление движения. Сначала мы определяем, нажата ли клавиша, а затем определяем ее код, записываем его в переменную ***key***, и обрабатываем это нажатие с помощью оператора `switch`.

```
#include <graphics.h>
#include <conio.h>
#include <dos.h>
void Figure ( int x, int y, int color )
{
    ... // здесь записываем ту же самую функцию, что и раньше
}
void main()
{
    int d = DETECT, m;
    int x, y, dx, dy, key;
    initgraph ( &d, &m, "c:\\borlandc\\bgi" );
    x = 320; y = 240;
    dx = 1; dy = 0;
    while ( 1 )
    {
```

← начать с центра экрана,
двигаться влево

```
        if ( kbhit() ) {
            key = getch();
            if ( key == 27 ) break;
            switch ( key ) {
                case 75: dx = - 1; dy = 0; break;
                case 77: dx = 1; dy = 0; break;
                case 72: dx = 0; dy = - 1; break;
                case 80: dx = 0; dy = 1;
            }
        }
```

← ЭТОТ БЛОК ВЫПОЛНЯЕТСЯ
ТОЛЬКО ТОГДА, КОГДА НАЖАТА
КАКАЯ-НИБУДЬ КЛАВИША

```
        Figure ( x, y, YELLOW );
        delay ( 10 );
        Figure ( x, y, BLACK );
```

```
        x += dx;
        y += dy;
```

```
    }
    closegraph();
}
```

11. Случайные и псевдослучайные числа

Что такое случайные числа?

Представьте себе снег, падающий на землю. Допустим, что мы сфотографировали природу в какой-то момент. Сможем ли мы ответить на такой вопрос: куда точно упадет следующая снежинка? Навряд ли, потому что это зависит от многих причин — от того, какая снежинка ближе к земле, как подует ветер и т.п. Можно сказать, что снежинка упадет в *случайное место*, то есть в такое место, которое нельзя предсказать заранее.

Для моделирования случайных процессов (типа падения снежинок, броуновского движения молекул вещества и т.п.) на компьютерах применяют *случайные числа*.

Случайные числа - это такая последовательность чисел, в которой невозможно назвать следующее число, зная сколько угодно предыдущих.

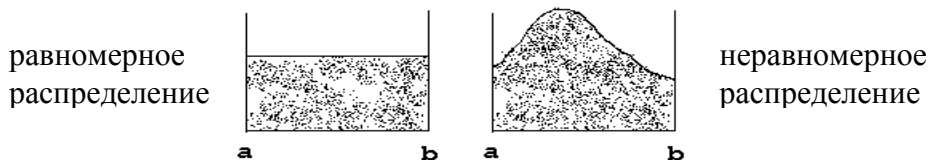
Получить случайные числа на компьютере достаточно сложно. Иногда для этого применяют различные источники радиошумов. Однако математики придумали более универсальный и удобный способ — *псевдослучайные числа*.

Псевдослучайные числа - это последовательность чисел, обладающая свойствами, близкими к свойствам случайных чисел, в которой каждое следующее число вычисляется на основе предыдущих по специальной математической формуле.

Таким образом, эта последовательность ведет себя так же, как и последовательность случайных чисел, хотя, зная формулу, мы можем получить следующее число в последовательности.

Распределение случайных чисел

Обычно используют псевдослучайные числа (далее для краткости мы называем их просто случайными), находящиеся в некотором интервале. Например, представим себе, что снежинки падают не на всю поверхность земли, а на отрезок оси *OX* от *a* до *b*. При этом очень важно знать некоторые *общие свойства* этой последовательности. Если понаблюдать за снежинками в безветренную погоду, то слой снега во всех местах будет примерно одинаковый, а при ветре — разный. Про случайные числа в первом случае говорят, что они имеют *равномерное распределение*, а во втором случае — *неравномерное*.



Большинство стандартных датчиков псевдослучайных чисел (то есть формул, по которым они вычисляются) дают равномерное распределение в некотором интервале.

Поскольку случайные числа в компьютере вычисляются по формуле, то для того, чтобы повторить в точности какую-нибудь случайную последовательность достаточно просто взять то же самое начальное значение.

Функции для работы со случайными числами

В языке Си существуют следующие функции для работы со случайными числами (их описание находится в заголовочном файле **stdlib.h** — это значит, что его необходимо подключать в начале программы):

<code>n = rand();</code>	получить случайное целое число в интервале от 0 до RAND_MAX (это очень большое целое число — 32767)
<code>n = random (max);</code>	получить случайное целое число в интервале от 0 до max-1
<code>srand (m);</code>	установить новое начальное значение случайной последовательности
<code>randomize();</code>	установить случайное начальное значение случайной последовательности



Случайные числа в заданном интервале

Для практических задач надо получать случайные числа в заданном интервале $[a, b]$. Важно уметь получать последовательности как целых, так и вещественных случайных чисел. С помощью случайных чисел решаются очень многие задачи, которые очень трудно (или невозможно) решить другими способами. Например, так моделируется движение молекул в физике и биологии.

Рассмотрим сначала целые числа. Если `random(n)` генерирует случайное число в интервале $[0, n-1]$, то очевидно, что

```
k = random(n) + a;
```

дает последовательность в интервале $[a, a+n-1]$. Поскольку нам нужно получить интервал $[a, b]$, мы имеем $b=a+n-1$, откуда $n=b-a+1$. Поэтому

Для получения случайных целых чисел с равномерным распределением в интервале $[a, b]$ надо использовать формулу

```
k = random(b-a+1) + a;
```

Более сложным оказывается вопрос о случайных вещественных числах. Самое лучшее, что мы можем сделать, это разделить результат функции `rand()` на `RAND_MAX`:

```
x = (float) rand() / RAND_MAX;
```

и получить таким образом, случайное вещественное число в интервале $[0, 1]$ (при этом надо не забыть привести одно из этих чисел к вещественному типу, иначе деление одного целого числа на большее целое число будет всегда давать ноль).

Длина интервала такой последовательности равна **1**, а нам надо получить в конечном счете интервал длиной **b-a**. Поэтому если теперь это число умножить на **b-a** и добавить к результату **a**, мы получаем как раз нужный интервал.

Для получения случайных вещественных чисел с равномерным распределением в интервале $[a, b]$ надо использовать формулу

```
x = rand() * (b-a) / RAND_MAX + a;
```

До этого момента мы говорили только о получении случайных чисел с равномерным распределением. Как же получить неравномерное? На этот вопрос математика отвечает так: из равномерного распределения можно получить неравномерное, применив к этим данным

некоторую математическую операцию. Например, чтобы основная часть чисел находилась в середине интервала, можно брать среднее арифметическое нескольких последовательных случайных чисел с равномерным распределением.



Снег на экране

Приведенная ниже программа генерирует случайное значение x в интервале $[160, 480]$, случайное значение y в интервале $[120, 360]$ и проверяет цвет точки с координатами (x, y) . Если эта точка черная, то ее цвет устанавливается случайный, а если нет — черный. Таким образом, разноцветный снег падает в центральный прямоугольник.

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

void main()
{
    int  d = VGA, m = VGAHI;
    int  x, y, color;
    initgraph ( &d, &m, "C:\\\\BORLANDC\\\\BGI" );
```

```
while ( ! kbhit () ) {
    x = random(320)+160;
    y = random(240)+120;
    color = random(16);
```

цикл: пока не нажата клавиша

получить случайные
координаты и цвет точки

```
if ( getpixel ( x, y ) != BLACK )
    putpixel ( x, y, BLACK );
else
    putpixel ( x, y, color );
}
getch();
closegraph();
}
```

проверить цвет точки



Что новенького?

- для определения того, была ли нажата какая-нибудь клавиша, используется функция `kbhit()`, которая возвращает 0, если клавиша не была нажата, и 1, если нажата клавиша. Для того, чтобы определить код этой клавиши, надо вызвать функцию `getch()`. Таким образом, цикл "пока не нажата клавиша" может выглядеть так:

```
while ( ! kbhit() ) { ... }
```

- для того, чтобы получить текущий цвет точки, используется функция `getpixel(x, y)`

Глава II.

Хранение и обработка данных

Глава II. Хранение и обработка данных	1
1. Массивы	2
Основные понятия	2
Ввод с клавиатуры и вывод на экран	3
Заполнение случайными числами	4
Работа с текстовыми файлами	6
Работа с двоичными файлами	9
Простой поиск в массиве	10
Перестановка элементов массива	14
Сортировка массивов	16
Двоичный поиск в массиве	19
Массивы в процедурах и функциях	20
2. Символьные строки	22
Что такое символьная строка?	22
Стандартный ввод и вывод	23
Работа с файлами	24
Функции для работы со строками	26
Строки в функциях и процедурах	37
3. Матрицы (двухмерные массивы)	39
Что такое матрица?	39
Объявление матриц	40
Стандартный ввод и вывод	40
Работа с файлами	42
Алгоритмы для работы с матрицами	43
4. Массивы символьных строк	46
Объявление и инициализация	46
Ввод и вывод	46
Сортировка	47
5. Управление памятью	48
Указатели	48
Динамическое выделение памяти	49
Выделение памяти для матрицы	51
6. Рекурсия	54
Что такое рекурсия?	54
Не допустим бесконечную рекурсию!	55
Когда рекурсия не нужна	56
Рекурсивный поиск	57
Рекурсивные фигуры	58
Перебор вариантов	59
Быстрая сортировка	62
7. Структуры	65
Что такое структуры?	65
Объявление и инициализация	65
Работа с полями структуры	66
Ввод и вывод	66
Копирование	67
Массивы структур	68
Динамическое выделение памяти	69
Структуры как параметры процедур	70
Сортировка по ключу	71

1. Массивы



Основные понятия



Что такое массив?

Основное предназначение компьютеров не вычисления, как считают многие, а **обработка больших объемов данных**. При размещении большого количества данных в памяти возникает такая проблема: надо научиться обращаться к каждой ячейке с данными отдельно. При этом очень сложно дать каждой ячейке собственное имя и при этом не запутаться. Выкручиваются из этой ситуации так: дают имя не ячейке, а группе ячеек, в которой каждая ячейка имеет номер. Такая область памяти называется массивом.

Массив - это группа ячеек памяти одинакового типа, расположенных рядом и имеющих общее имя. Каждая ячейка в группе имеет уникальный номер.

При работе с массивами надо научиться решать три задачи:

- выделять память нужного размера под массив
- записывать данные в нужную ячейку
- читать данные из ячейки



Объявление массива

Чтобы использовать массив, надо его объявить – выделить место в памяти. Типом массива называется тип массива это тип входящих в него элементов. Массивы могут быть разных типов — `int`, `float`, `char`, и т.д. Массив объявляют так же, как и обычные переменные, но после имени массива в квадратных скобках записывается его размер.

```
int A[10], B[20];    // 2 массива на 10 и 20 целых чисел
float C[12];         // массив из 12 вещественных чисел
```

При объявлении массива можно сразу заполнить его начальными значениями, перечисляя их внутри фигурных скобок:

```
int A[4] = { 2, 3, 12, 76 };
```

Если в списке в фигурных скобках записано меньше чисел, чем элементов в массиве, то оставшиеся заполняются нулями. Если чисел больше, чем надо, транслятор сообщает об ошибке. Например,

```
int A[4] = { 2 }; // при этом последние три элемента равны 0
```

Для повышения универсальности программы размер массива лучше определять в директиве препроцессора `#define`, например

```
#define N 10
```

означает заменить во всем тексте программы слово **N** на число **10**. В этом случае для переделки программы для массива другого размера надо только поменять число в строке `#define`. Пример:

```
#define N 20
void main()
{
    int A[N];
    ...
}
```


В современных программах на языке Си++ рекомендуется использовать константы вместо директивы `#define`, например

```
const N = 20;

void main()
{
    int A[N];
    ...
}
```

В отличие от предыдущего варианта, здесь надо ставить знак «равно» и точку с запятой в конце оператора.

В таблице показаны примеры правильного и неправильного объявления массива.

правильно		неправильно	
<code>int A[20];</code>	размер массива указан явно	<code>int A[];</code>	размер массива неизвестен
<code>#define N 20</code> <code>int A[N];</code>	размер массива – постоянная величина	<code>int N = 20;</code> <code>int A[N];</code>	размер массива не может быть переменной
<code>const N = 20;</code> <code>int A[N];</code>	размер массива – постоянная величина		



Обращение к элементу массива

Каждый элемент массива имеет свой порядковый номер. Чтобы обратиться к элементу массива, надо написать имя массива и затем в квадратных скобках номер нужного элемента. Важно запомнить одно важное правило:

Элементы массивов в языке Си нумеруются с нуля. таким образом, если в массиве 10 элементов, он содержит элементы:

`A[0], A[1], A[2], ..., A[9]`

Номер элемента массива называется его *индексом*. Вот примеры обращения к массиву **A**:

```
x = (A[3] + 5) * A[1]; // прочитать значения A[3] и A[1]
A[0] = x + 6;         // записать новое значение в A[0]
```

В языке Си не контролируется **выход за границы массива**, то есть формально вы можете записать что-то в элемент с несуществующим индексом, например в **`A[345]`** или в **`A[-12]`**. Однако при этом вы стираете какую-то ячейку в памяти, не относящуюся к массиву, поэтому последствия такого шага непредсказуемы и во многих случаях программа зависает.



Ввод с клавиатуры и вывод на экран

Как же ввести данные в массив? Существует много способов в зависимости от вашей задачи:

- элементы массива вводятся с клавиатуры вручную
- массив заполняется случайными числами (например, для моделирования случайных процессов)
- элементы массива читаются из файла (текстового или двоичного)
- элементы массива поступают через порт с внешнего устройства (например, сканера, модема и т.п.)
- массив заполняется в процессе вычислений

Задача. Ввести с клавиатуры массив из 10 элементов, умножить все элементы на 2 и вывести полученный массив на экран.

К сожалению, невозможно просто сказать компьютеру: «введи массив **A**». Мы должны каждый элемент прочесть отдельно.

Чтобы ввести массив в память, надо каждый его элемент обработать отдельно (например, вызвав для него функцию ввода `scanf`).

Ввод с клавиатуры применяется в простейших программах, когда объем вводимой информации невелик. Для ввода массива будем использовать цикл `for`. Напомним, что массив надо предварительно **объявить**, то есть выделить под него память.

Вводить можно столько элементов массива, сколько ячеек памяти выделено. Помните, что элементы массива нумеруются с нуля, поэтому если массив имеет всего 10 элементов, то последний элемент имеет номер 9. Если пытаться записывать в 10-ый элемент, произойдет выход за границы массива и программа скорее всего будет работать неверно (а может быть и зависнет). При вводе массива желательно выдать на экран общую подсказку для ввода всего массива и подсказки для каждого элемента.

Для умножения элементов массива на 2 надо снова использовать цикл, в котором за один раз обрабатывается 1 элемент массива.

Вывод массива на экран выполняется также в цикле `for`. Элементы выводятся по одному. Если в конце строки-формата в операторе `printf` поставить пробел, то элементы массива будут напечатаны в строку, а если символ `"\n"` – то в столбик.

<code>#include <stdio.h></code>	
<code>const N = 10;</code>	размер массива
<code>void main()</code>	
<code>{</code>	
<code>int i, A[N];</code>	объявить массив
<code>printf("Введите массив A\n");</code>	подсказка для ввода
<code>for (i = 0; i < N; i ++) {</code>	
<code> printf("Введите A[%d]> ", i);</code>	
<code> scanf ("%d", &A[i]);</code>	ввод элемента массива с номером i
<code>}</code>	
<code>for (i = 0; i < N; i ++)</code>	
<code> A[i] = A[i] * 2;</code>	умножить элементы массива на 2
<code>printf("\nРезультат:\n");</code>	
<code>for (i = 0; i < N; i ++)</code>	
<code> printf("%d ", A[i]);</code>	вывод элемента массива с номером i
<code>}</code>	



Заполнение случайными числами

Этот прием используется для моделирования случайных процессов, например, броуновского движения частиц. Пусть требуется заполнить массив равномерно распределенными случайными числами в интервале $[a, b]$. Поскольку для целых и вещественных чисел способы вычисления случайного числа в заданном интервале отличаются, рассмотрим оба варианта. Здесь и далее предполагается, что в начале программы есть строка

```
#define N 10
```

или

```
const N = 10;
```

Как вы уже знаете из первой части курса, для получения случайных чисел с равномерным распределением в интервале $[a, b]$ надо использовать формулу

```
k = random ( b - a + 1 ) + a;
```

Для вещественных чисел формула несколько другая:

```
x = rand() * (b - a) / RAND_MAX + a;
```

В приведенном ниже примере массив **A** заполняется случайными целыми числами в интервале $[-5, 10]$, а массив **X** – случайными вещественными числами в том же интервале.

```
#include <stdlib.h>
const N = 10;
void main()
{
    int i, A[N], a = -5, b = 10;
    float X[N];

    for ( i = 0; i < N; i ++ )
        A[i] = random(b-a+1) + a;
    for ( i = 0; i < N; i ++ )
        X[i] = (float)rand() * (b-a) / RAND_MAX + a;
    ...
}
```

Возможно, в этом примере не вполне ясно, зачем перед вызовом функции `rand()` поставлено слово `(float)`. Это связано с тем, что у нас **a** и **b** – целые числа. Результат функции `rand()` – тоже целое число. Здесь возможны две проблемы:

- При умножении результата функции `rand()` на $(b - a)$ может получиться очень большое число, которое не поместится в переменную типа `int`.
- В языке Си при делении целого числа на целое остаток отбрасывается, поэтому при делении результат будет неверным.

Когда массив заполняется случайными числами, обязательно вывести на экран исходный массив.

Задача. Заполнить массив случайными целыми числами в интервале $[-10, 15]$, умножить все элементы на 2 и вывести на экран исходный массив и результат.

```
#include <stdio.h>
#include <stdlib.h>
const N = 10;
void main()
{
    int i, A[N];

    for ( i = 0; i < N; i ++ )
        A[i] = random(26) - 10;

    printf("\n Исходный массив:\n");
    for ( i = 0; i < N; i ++ )
        printf("%d ", A[i]);

    for ( i = 0; i < N; i ++ )
        A[i] = A[i] * 2;

    printf("\n Результат:\n");
    for ( i = 0; i < N; i ++ )
        printf("%d ", A[i]);
}
```



Работа с текстовыми файлами

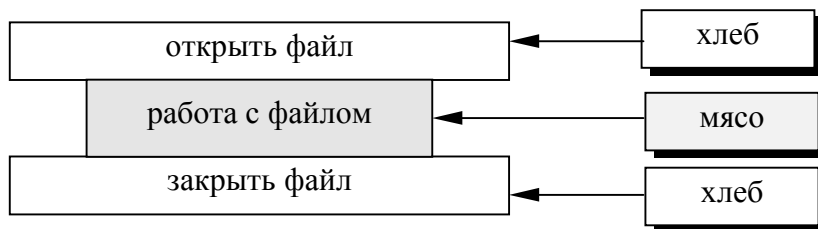
Наверняка при работе с предыдущими программами вы ощутили, что вводить с клавиатуры большое количество данных очень утомительно, особенно если это надо делать много раз. В таких случаях создают файл на диске, в который записывают нужные данные, а программа сама читает данные из файла.

Файлы бывают текстовые (в которых можно записывать только буквы, цифры, скобки и т.п.) и двоичные (в которых могут храниться любые символы из таблицы). В этом разделе мы рассмотрим только текстовые файлы.



Как работать с файлами из программы

Работа с файлами, так же как и работа в графическом режиме, строит по принципу сэндвича:



Понятие «открыть файл» означает «начать с ним работу», сделать его активным и заблокировать обращение других программ к этому файлу. При закрытии файла он освобождается (теперь с ним могут работать другие программы) и все ваши изменения вносятся на диск.

Для работы с файлом используется специальная переменная, которая называется *указателем на файл*. Это адрес блока данных в памяти, в котором хранится вся информация об открытом файле. Объявляется указатель на файл так:

```
FILE *fp;
```

Чтобы открыть файл, надо вызвать функцию `fopen`, которая попытается открыть файл и записать его адрес в переменную ***fp***. После этого все обращения к файлу выполняются не по имени файла, а через указатель ***fp***.

```
fp = fopen ( "qq.dat", "r" );
```

Здесь файл `"qq.dat"` из текущего каталога открывается для чтения (режим `"r"` во втором параметре функции `fopen`). Если надо, можно указать полный (или относительный) путь к файлу, например так:

```
fp = fopen ( "c:\\data\\qq.dat", "r" );
```

Знак «наклонные черта» (слэш) в символьных строках всегда удваивается, потому что одиночный слэш – это специальный символ, например в сочетании `"\\n"`.

Кроме режима `"r"` (чтение из файла) есть еще два основных режима:

- `"w"` Запись в новый файл. Если на диске уже есть файл с таким именем, он будет предварительно удален.
- `"a"` Добавление в конец файла. Если на диске уже есть файл с таким именем, новые данные дописываются в конец файла. Если такого файла нет, то он будет создан.
- `"r+"` Открыть существующий файл для изменения с возможностью записи и чтения

"w+" Создание нового файла для записи и чтения (если файл с таким именем уже есть, он заменяется новым).

Иногда программа не может открыть файл. Если файл открывается на чтение, это возможно в следующих случаях:

- неверно задано имя файла или файла нет на диске
- файл используется другой программой и заблокирован

Если файл открывается на запись, операция может закончиться неудачно, если

- на диске нет места
- файл защищен от записи
- неверно задано имя файла (например, оно содержит две точки, знак вопроса и т.п.)

Если файл не удалось открыть, функция `fopen` возвращает специальное нулевое значение (*нулевой указатель*), который обозначается **NULL**. Поэтому надо всегда проверять правильность открытия файла, особенно в режиме чтения. Если файл не был открыт, надо вывести сообщение об ошибке и выйти из программы.

```
if ( fp == NULL )
{
    printf("Нет файла с данными");
    return;
}
```

Если файл открыт, можно читать из него данные. Для того используем функцию `fscanf`. Она полностью аналогична `scanf`, но служит для ввода из файла, а не с клавиатуры. Кроме того, ее первый параметр – указатель на файл, из которого читаются данные.

```
n = fscanf ( fp, "%d", &A[i] );
```

Функция `fscanf` возвращает результат – количество чисел, которые ей удалось прочесть. Если мы запрашивали одно число, то значение переменной **n** может быть равно единице (если все нормально) или нулю (если данные закончились или ошибочные, например, вместо чисел введено слово). Для успешного чтения данные в файле должны отделяться пробелом или символом перехода на новую строку (он вставляется в файл при нажатии на клавишу **Enter**).

Если файл открыт на запись, можно записать в него данные с помощью функции `fprintf`, которая полностью аналогична `printf`.

Когда работа с файлом закончена, надо закрыть его, вызвав функцию `fclose`:

```
fclose ( fp );
```



Массив известного размера

Задача. Ввести массив из 10 целых чисел из файла **"input.dat"**, умножить каждый элемент на 2 и вывести в столбик в файл **"output.dat"**.

Эта задача решается с помощью функций `fopen`, `fscanf`, `fprintf` и `fclose`, описанных выше. В программе обрабатываются две ошибки:

- файла нет (его не удалось открыть)
- в файле мало данных или данные неверные (например, слова вместо целых чисел)

```

#include <stdio.h>
void main()
{
    int i, A[N];
    FILE *fp;
    fp = fopen( "input.dat", "r" );
    if ( fp == NULL ) {
        printf("Нет файла данных");
        return;
    }
    for ( i = 0; i < N; i ++ )
        if ( 0 == fscanf(fp, "%d", &A[i]) ) {
            printf("Не хватает данных в файле");
            break;
        }
    fclose ( fp );
    for ( i = 0; i < N; i ++ )
        A[i] = A[i] * 2;
    fp = fopen( "output.dat", "w" );
    for ( i = 0; i < N; i ++ )
        fprintf ( fp, "%d\n", A[i] );
    fclose ( fp );
}

```

указатель на файл

открыть файл на чтение

чтение и обработка ошибки

заккрыть файл

запись массива в файл

В отличие от предыдущих, эта программа выдает результаты не на экран, а в файл **"output.dat"** в текущем каталоге.



Массив неизвестного размера

Задача. В файле **"input.dat"** записаны в два столбика пары чисел ***x*** и ***y***. Записать в файл **"output.dat"** в столбик суммы ***x+y*** для каждой пары.

Сложность этой задачи состоит в том, что мы не можем прочитать все данные сразу в память, обработать их и записать в выходной файл. Не можем потому, что не знаем, сколько пар чисел в массиве. Конечно, если известно, что в файле, скажем, не более 200 чисел, можно выделить массив «с запасом», прочитать столько данных, сколько нужно, и работать только с ними. Однако в файле могут быть миллионы чисел и такие массивы не поместятся в памяти.

Решение задачи основано на том, что для вычисления суммы каждой пары нужны только два числа, а остальные мы можем не хранить в памяти. Когда вычислили их сумму, ее не надо хранить в памяти, а можно сразу записать в выходной файл. Поэтому будем использовать такой алгоритм

1. Открыть два файла, один на чтение (с исходными данными), второй – на запись.
2. Попытаться прочитать два числа в переменные ***x*** и ***y***; если это не получилось (нет больше данных или неверные данные), закончить работу.
3. Сложить ***x*** и ***y*** и записать результат в выходной файл.
4. Перейти к шагу 2.

Для того, чтобы определить, удачно ли закончилось чтение, мы будем использовать то, что функция `scanf` возвращает количество удачно считанных чисел. Мы будем читать за один раз сразу два числа, ***x*** и ***y***. Если все закончилось удачно, функция `fscanf` возвращает значение 2 (обе переменных прочитаны). Если результат этой функции меньше двух, данные закончились или неверные.

Заметим, что надо работать одновременно с двумя открытыми файлами, поэтому в памяти надо использовать два указателя на файлы, они обозначены именами ***fin*** и ***fout***. Для сокращения записи ошибки при открытии файлов не обрабатываются.

#include <stdio.h>	
void main()	
{	
int n, x, y, sum;	указатели на файлы
FILE *fin, *fout;	открыть файл на чтение
fin = fopen("input.dat", "r");	открыть файл на запись
fout = fopen("output.dat", "w");	
while (1) {	
n = fscanf ("%d%d", &x, &y);	
if (n < 2) break;	данные закончились
sum = x + y;	
fprintf ("%d\n", sum);	
}	
fclose (fout);	закрывать файлы
fclose (fin);	
}	

В программе используется бесконечный цикл `while`. Программа выходит из него тогда, когда данные в файле закончились.



Работа с двоичными файлами

Двоичные файлы отличаются от текстовых тем, что в них записана информация во внутреннем машинном представлении. Двоичный файл нельзя просмотреть на экране (вернее, можно просмотреть, но очень сложно понять). Но есть и преимущества - из двоичных файлов можно читать сразу весь массив в виде единого блока. Также можно записать весь массив или его любой непрерывный кусок за одну команду.

При открытии двоичного файла вместо режимов `"r"`, `"w"` и `"a"` используют соответственно `"rb"`, `"wb"` и `"ab"`. Дополнительная буква `"b"` указывает на то, что файл двоичный (от английского слова *binary* - двоичный). Решение предыдущей задачи, использующее двоичный файл, показано ниже.

```

#include <stdio.h>
void main()
{
    int i, n, A[N];
    FILE *fp;
    fp = fopen( "input.dat", "rb" );
    n = fread ( A, sizeof(integer), N, fp );
    if ( n < N ) {
        printf("Не хватает данных в файле");
        break;
    }
    fclose ( fp );
    for ( i = 0; i < N; i ++ )
        A[i] = A[i] * 2;
    fp = fopen( "output.dat", "wb" );
    fwrite ( A, sizeof(integer), N, fp );
    fclose ( fp );
}

```

указатель на файл

открыть двоичный файл на чтение

читаем весь массив сразу

обработка ошибки

закреть файл

запись массива в файл

Для чтения из двоичного файла используется функция `fread`, которая принимает 4 параметра:

- Адрес области в памяти, куда записать прочитанные данные (в данном случае это адрес первого элемента массива **A**, который обозначается как **&A[0]** или просто **A**).
- Размер одного элемента данных (лучше сделать так, чтобы машина сама определила его, например, в нашем случае - `sizeof(integer)` - размер целого числа. Хотя в *Borland C 3.1* целое число занимает 2 байта, во многих современных системах программирования его размер - 4 байта. Наша программа будет работать и в этом случае, то есть станет *переносимой* на другую платформу).
- Количество элементов данных в массиве (**N**).
- Указатель на открытый файл, откуда читать данные (**fp**).

Функция `fread` возвращает количество успешно прочитанных элементов массива - ее возвращаемое значение можно использовать для обработки ошибок. Если функция `fread` вернула значение, меньшее, чем **N**, в файле не хватает данных.

Для записи массива в двоичный файл используется функция `fwrite` с такими же параметрами; она возвращает количество успешно записанных элементов.

Преимущество этого способа состоит в том, что массив читается и записывается сразу единым блоком. Это значительно увеличивает скорость записи на диск (в сравнении с выводом в текстовый файл отдельно каждого элемента).



Простой поиск в массиве

Во многих задачах требуется последовательно перебрать все элементы массива и найти нужные нам. Мы рассмотрим четыре таких задачи:

- поиск одного заданного элемента
- вывод всех элементов, которые удовлетворяют заданному условию
- формирование нового массива из всех отобранных элементов
- поиск минимального (максимального) элемента

Все эти задачи решаются с помощью цикла, в котором перебираются все элементы массива от начального (0-ого) до конечного ($N-1$ -ого) элемента. Такой поиск называется *линейным*, поскольку все элементы просматриваются последовательно один за другим.



Поиск одного элемента

Задача. Определить, есть ли в массиве элемент с заданным значением x , и если он есть - найти его номер.

Если нет никакой информации о разыскиваемых данных, то применяется линейный поиск, основная идея которого - последовательно просматривать массив, пока не будет обнаружено совпадение или достигнут конец массива. Это реализует следующая простая программа:

```
#include <stdio.h>
const N = 10;
void main()
{
    int i, A[N];
    int success = 0;

    // ввод массива
    for ( i = 0; i < N; i ++ )
        if ( A[i] == x ) {
            success = 1;
            break;
        }

    if ( success )
        printf ( "Индекс нужного элемента %d", i );
    else
        printf ( "Заданный элемент не найден" );
}
```

переменная - флаг

нашли то, что надо

Чтобы определить ситуацию, когда элемент не найден, нам надо ввести специальный флаг **success**, который устанавливается в 1, если элемент найден. Таким образом, если после цикла **success = 0**, то в массиве нет нужного элемента. Напомним, что значение переменной цикла **i** для этого использовать нельзя, потому что оно не определено за пределами цикла в соответствии с правилами языка Си.

Для линейного поиска в худшем случае мы имеем N сравнений. Понятно, что для ускорения поиска надо сначала как-то упорядочить данные, в этом случае можно сделать поиск эффективным.



Поиск всех элементов, соответствующих условию

Задача. Определить, сколько в массиве положительных элементов и вывести их на экран.

Для решения этой задачи вводим *счетчик* – специальную переменную, значение которой будет увеличиваться на единицу, когда мы нашли очередной положительный элемент.

```

#include <stdio.h>
const N = 10;
void main()
{
    int i, A[N], count = 0;
    // ввод массива

    for ( i = 0; i < N; i ++ )
        if ( A[i] > 0 ) {
            count ++;
            printf ("%d ", A[i]);
        }

    printf ("\n В массиве %d положительных чисел", count);
}

```

счетчик

нашли положительный

Формирование массива по заданному условию

Задача. Сформировать новый массив **B**, включив в него все положительные элементы исходного массива **A**, и вывести его на экран.

Пусть есть массив **A[N]**. Надо выбрать из него все положительные элементы и записать их в новый массив, который и будет дальше использоваться.

Сначала надо определить, сколько места в памяти надо выделить для массива **B**. В “худшем” случае все элементы в массиве **A** будут положительными и войдут в массив **B**, поэтому массив **B** должен иметь такой же размер, что и массив **A**.

Можно предложить такой способ: просматривать весь массив **A**, и если $A[i] > 0$, переписать его значение в **B[i]**.

A

1	-1	3	-2	5
---	----	---	----	---

 \longrightarrow B

1	?	3	?	5
---	---	---	---	---

В этом случае использовать массив **B** очень сложно, потому что нужные элементы стоят не подряд.

Более красивый способ состоит в следующем. Объявляем временную переменную-счетчик **count**, в которой будем хранить количество найденных положительных элементов. Сначала она равна нулю. Если нашли очередной положительный элемент, то ставим его в ячейку **B[count]** и увеличиваем счетчик. Таким образом, все нужные элементы стоят в начале массива **B**.

A

1	-1	3	-2	5
---	----	---	----	---

 \longrightarrow B

1	3	5	?	?
---	---	---	---	---

```

#include <stdio.h>
const N = 10;
void main()
{
    int i, A[N], B[N], count = 0;
    // ввод массива A

    for ( i = 0; i < N; i ++ )
        if ( A[i] > 0 ) {
            B[count] = A[i];
            count ++;
        }

    printf("\n Результат:\n");
    for ( i = 0; i < count; i ++ )
        printf("%d ", B[i]);
}

```

выбираем положительные числа в массив **B**

можно сделать и так:
 if (A[i] > 0)
 B[count++] = A[i];

вывод результата

Обратите внимание, что переменная в последнем цикле изменяется от **1** до **count**, так что на экран выводятся только реально используемые элементы массива **B**.



Минимальный элемент

Задача. Найти и вывести на экран минимальный элемент в массиве **A**.

Для решения задачи надо выделить в памяти ячейку (переменную) для хранения найденного минимального значения. Сначала мы записываем в эту ячейку первый элемент. Затем смотрим следующий элемент и сравниваем его с минимальным. Если он меньше минимального, записываем его значение в ячейку минимального элемента. И так далее. Когда мы рассмотрим последний элемент в массиве, в дополнительной ячейке памяти будет минимальное значение из всех элементов массива.

Заметим, что перебор в цикле начинается с элемента с номером 1, поскольку начальный элемент мы рассмотрели отдельно.

```

#include <stdio.h>
const N = 10;
void main()
{
    int i, A[N], min;
    // ввод массива A

    min = A[0];
    for ( i = 1; i < N; i ++ )
        if ( A[i] < min )
            min = A[i];

    printf("\n Минимальный элемент %d", min);
}

```

предполагаем, что **A[0]** - минимальный

перебор с элемента 1

Чтобы найти максимальный элемент, достаточно изменить условие в заголовке условного оператора на обратное ($A[i] > \text{min}$). Конечно, вспомогательную переменную в этом случае лучше (но не обязательно!) назвать **max**.

Теперь можно усложнить задачу и найти еще и номер минимального элемента.

Задача. Найти и вывести на экран минимальный элемент в массиве **A** и его номер.

Напрашивается такое решение: завести еще одну переменную, в которой хранить номер минимального элемента. Если мы нашли новый минимальный элемент, то в одну переменную записали его значение, а во вторую - его номер.

Тем не менее, можно обойтись одной дополнительной переменной. Как? Дело в том, что по номеру элемента можно легко найти его значение в массиве. На этом основана приведенная ниже программа. Теперь мы запоминаем (в переменной **nMin**) не значение минимального элемента, а только его номер.

```
#include <stdio.h>
const N = 10;
void main()
{
    int i, A[N], nMin;
    // ввод массива A

    nMin = 0;
    for ( i = 1; i < N; i ++ )
        if ( A[i] < A[nMin] )
            nMin = i;

    printf("\n Минимальный элемент A[%d]=%d", nMin, A[nMin]);
}
```

номер минимального элемента

запомнили новый номер



Перестановка элементов массива



О кувшине и вазе

Представьте, что в вазе для цветов налито молоко, а в кувшине - вода с удобрениями. Как привести все в порядок? Надо использовать третью емкость такого же (или большего) объема. Сначала переливаем в нее воду из кувшина (или молоко из вазы, все равно), затем в пустой кувшин переливаем молоко (или в вазу - воду), а затем из третьей емкости переливаем воду в вазу (или, соответственно, молоко в кувшин).

Так же и в программировании: чтобы поменять местами значения двух ячеек в памяти, надо использовать временную переменную¹. Пусть даны ячейки **a** и **b**, содержащие некоторые значения. После выполнения следующих команд их значения поменяются:

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

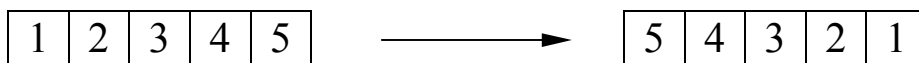
Эта цепочка операторов присваивания особая:

- она начинается и заканчивается временной переменной **c**
- следующий оператор начинается с той переменной, на которую закончился предыдущий



Перестановка наоборот (инверсия)

Инверсия (от английского *inverse* - обратный) - это такая перестановка, когда первый элемент становится последним, второй - предпоследним и т.д.



¹ Существуют и немного более сложные способы, позволяющие обойтись всего двумя ячейками

Эта простая задача имеет один подводный камень. Пусть размер массива N . Тогда элемент $A[0]$ надо переставить с $A[N-1]$, $A[1]$ с $A[N-2]$ и т.д. Заметим, что в любом случае сумма индексов переставляемых элементов равна $N-1$, поэтому хочется сделать цикл от 0 до $N-1$, в котором переставить элементы $A[i]$ с $A[N-1-i]$. Однако при этом вы обнаружите, что массив не изменился.

Обратим внимание, что перестановка затрагивает одновременно и первую, и вторую половину массива. Поэтому сначала все элементы будут переставлены правильно, а затем (когда $i > N/2$) будут возвращены на свои места. Правильное решение - делать перебор, при котором переменная цикла доходит только до середины массива.

```
#include <stdio.h>

const N = 10;

void main()
{
    int i, A[N], c;

    // ввод массива A

    for ( i = 0; i < N/2; i ++ )
    {
        c = A[i];
        A[i] = A[N-1-i];
        A[N-1-i] = c;
    }

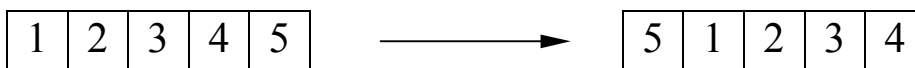
    printf("\n Результат:\n");
    for ( i = 0; i < N; i ++ )
        printf("%d ", A[i]);
}
```

перебор только до
середины массива



Циклический сдвиг

При циклическом сдвиге (вправо) первый элемент переходит на место второго, второй на место третьего и т.д., а последний элемент - на место первого.



Для выполнения циклического сдвига нам будет нужна временная переменная - в ней мы сохраним значение последнего элемента, пока будем переставлять остальные. Обратите внимание, что мы начинаем с конца массива, иначе массив просто заполнится первым элементом. Первый элемент ставится отдельно - копированием из временной переменной.

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

```
const N = 10;
```

```
void main()
```

```
{
  int i, A[N], c;
```

```
    // ввод массива A
```

```
c = A[N-1];
```

```
for ( i = N-1; i > 0; i -- )
    A[i] = A[i-1];
```

```
A[0] = c;
```

```
printf("\n Результат:\n");
```

```
for ( i = 0; i < N; i ++ )
    printf("%d ", A[i]);
```

```
}
```

запомнили последний элемент

цикл “в обратную сторону”:
i уменьшается

первый элемент ставим отдельно



Сортировка массивов

Сортировка - это расстановка элементов некоторого списка в заданном порядке

Существуют разные виды сортировки (по алфавиту, по датам и т.д.), они отличаются лишь процедурой сравнения элементов. Мы рассмотрим простейший вариант сортировки - расстановку элементов массива в порядке возрастания.

Программисты придумали множество методов сортировки. Они делятся на две группы:

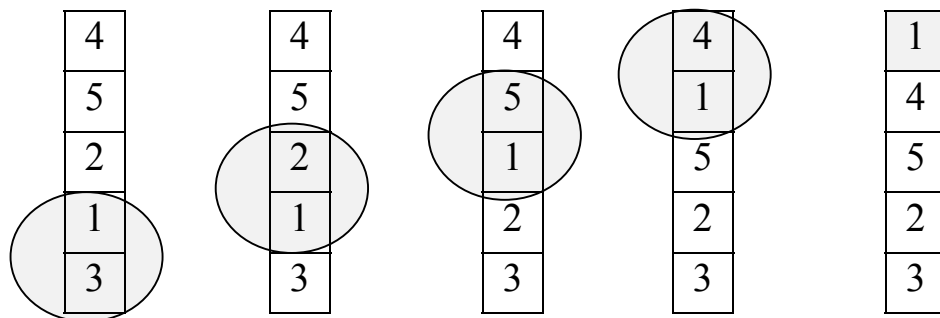
- понятные, но не эффективные
- эффективные, но непонятные (быстрая сортировка и т.п.).

Пока мы будем изучать только методы из первой группы, которых хватает для простых задач.



Метод пузырька

Название этого метода произошло от известного физического явления - пузырек воздуха в воде поднимается вверх. В этом методе сначала поднимается “наверх” (к началу массива) самый “легкий” элемент (минимальный), затем следующий и т.д. Делается это так. Сравниваем последний элемент с предпоследним. Если они стоят неправильно, то меняем их местами. Далее так же рассматриваем следующую пару элементов и т.д. Когда мы обработали пару **A[0]-A[1]**, минимальный элемент стоит на месте **A[0]**. Это значит, что на следующих этапах его можно не рассматривать



При следующем проходе наша задача - поставить на место элемент **A[1]**. Делаем это так же, но уже не рассматриваем **A[0]**, который стоит на своем месте. Сделав **N-1** проходов, мы

установим на место элементы **$A[0]$ – $A[N-2]$** . Это значит, что последний элемент уже тоже стоит на своем месте.

```
#include <stdio.h>
const N = 10;
void main()
{
    int i, j, A[N], c;
    // ввод массива A
    for ( i = 0; i < N-1; i ++ )
        for ( j = N-2; j >= i; j -- )
            if ( A[j] > A[j+1] )
            {
                c = A[j]; A[j] = A[j+1];
                A[j+1] = c;
            }
    printf("\n Отсортированный массив:\n");
    for ( i = 0; i < N; i ++ )
        printf("%d ", A[i]);
}
```

достаточно поставить на место **$N-1$** элементов

цикл “в обратную сторону”: **j** (номер верхнего элемента в паре) уменьшается, последняя пара (при **$j=i$**) будет **$A[i]$** и **$A[i+1]$**

меняем местами **$A[j]$** и **$A[j+1]$**

Метод пузырька работает медленно, особенно на больших массивах. Доказано, что при увеличении размера массива в 10 раз время выполнения программы увеличивается в 100 раз (метод имеет порядок **N^2**). К сожалению, все простые алгоритмы сортировки имеют такой (*квадратичный*) порядок.



Метод выбора минимального элемента

Еще один недостаток метода пузырька состоит в том, что приходится слишком часто переставлять местами соседние элементы. Этого можно избежать, если использовать *метод выбора минимального элемента*. Он заключается в следующем. Ищем в массиве минимальный элемент и ставим его на первое место. Затем из оставшихся элементов также ищем минимальный и ставим на следующее место и т.д.

В сравнении с методом пузырька, этот метод требует значительно меньше перестановок элементов (в худшем случае **$N-1$**). Он дает значительный выигрыш, если перестановки сложны и занимают много времени.

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

```
const N = 10;
```

```
void main()
```

```
{
  int i, j, nMin, A[N], c;
```

```
    // ввод массива A
```

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

```
    {
```

```
        nMin = A[i];
```

```
        for ( j = i+1; j < N-1; j ++ )
```

```
            if ( A[j] < A[nMin] )
```

```
                nMin = j;
```

```
        if ( nMin != i )
```

```
        {
```

```
            c = A[i]; A[i] = A[nMin];
```

```
            A[nMin] = c;
```

```
        }
```

```
    }
```

```
    printf("\n Отсортированный массив:\n" );
```

```
    for ( i = 0; i < N; i ++ )
```

```
        printf("%d ", A[i]);
```

```
}
```

достаточно поставить на место ***N-1*** элементов

ищем номер минимального элемент, начиная с ***A[i]*** до конца массива

меняем местами ***A[i]*** и ***A[nMin]***, если это надо



Метод попарного сравнения

Этот метод работает чуть-чуть быстрее (в **1.33** раза), чем метод пузырька. Еще одно его преимущество – он (в отличие от метода выбора минимального элемента) использует только одну операцию – упорядочивание двух элементов (так же, как и метод пузырька).

Предположим, что размер массива ***N*** - четное число. Тогда алгоритм сортировки заключается в следующем:

- упорядочим попарно соседние элементы (0 и 1, 2 и 3, 4 и 5 и т.д.) - ***N/2*** операций сравнения
- упорядочивая последовательно элементы (***N-2*** и ***N-4***, ***N-4*** и ***N-6***, ... 4 и 2, 2 и 0), то есть минимальные в парах, переведем на место ***A[0]*** минимальный элемент (***N/2*** операций)
- упорядочивая последовательно элементы (элементы **1** и **3**, **3** и **5**, ... ***N-5*** и ***N-3***, ***N-3*** и ***N-1***), то есть максимальные в парах, переведем на место ***A[N-1]*** максимальный элемент (***N/2*** операций)
- теперь первый и последний элементы стоят на своих местах и можно применить этот же алгоритм к оставшимся средним элементам

В алгоритме используется операция упорядочивания двух элементов, которую можно записать в виде процедуры


```
void Order ( int &a, int &b)
{
    int c;
    if ( a > b ) {
        c = a; a = b; b = c;
    }
}
```

Перед именем параметров процедуры стоит знак **&** - *передача по ссылке*. Это значит, что процедура меняет значения соответствующих ячеек в вызывающей программе. В процедуре мы меняем значения ячеек так, чтобы в первой из них было меньшее число.

В основной программе мы будем использовать две переменных - номера начального и конечного элемента неотсортированной части массива. Назовем их **from** и **to**. Сделаем цикл по переменной **from**: она будет меняться от **0** до **N/2-1**. Поскольку обе переменные меняются синхронно (к середине массива), зная **from** можно сразу найти, что **to = N - 1 - from**.

```
#include <stdio.h>
const N = 10;
void main()
{
    int i, from, to, A[N], c;
    // ввод массива A
    for ( from = 0; from < N/2; from ++ ) {
        to = N - 1 - from;
        for ( i = from; i < to; i += 2 )
            Order ( A[i], A[i+1] );
        for ( i = to-1; i > from; i -= 2 )
            Order ( A[i-2], A[i] );
        for ( i = from+1; i < to; i += 2 )
            Order ( A[i], A[i+2] );
    }

    printf("\n Отсортированный массив:\n");
    for ( i = 0; i < N; i ++ )
        printf("%d ", A[i]);
}
```

сортируем
соседние пары

ставим начальный
элемент **A[from]**

ставим последний
элемент **A[to]**

Обратите внимание, что во всех внутренних циклах переменная **i** изменяется с шагом 2 (через один).

Мы сказали, что алгоритм работает только для четных **N**, а можно ли использовать его для нечетных? Оказывается можно, если сначала поставить на место (любым способом) последний элемент, а затем отсортировать оставшуюся часть этим способом (ведь останется четное число неотсортированных элементов).



Двоичный поиск в массиве

Пусть элементы массива **A** уже расставлены по возрастанию и требуется найти элемент, равный **x**, среди элементов с номерами от **L** до **R**. Для этого использую следующую идею: выбираем средний элемент между **L** и **R**, он имеет номер $m = (L+R) / 2$, где деление выполняется нацело. Сравним его с искомым **x**. Если он равен **x**, поиск завершен. Если **x**

меньше элемента с номером, надо искать дальше между L и m , если больше m , дальше m и R . Обратите внимание, что элемент $A[R]$ не рассматривается.

```
#include <stdio.h>
const N = 10;
void main()
{
    int L = 0, R = N, m, A[N], flag = 0;
    // ввод массива A
    printf("Введите искомый элемент\n");
    scanf( "%d", x );
    while ( L < R ) {
        m = (L + R) / 2;
        if ( A[m] == x ) {
            flag = 1;
            break;
        }
        if ( x < A[m] ) R = m;
        else L = m + 1;
    }
    if ( flag )
        printf ( "Индекс нужного элемента %d", m );
    else printf ( "Такого элемента нет" );
}
```

нашли середину интервала

нашли элемент, равный x

сужаем границы

Переменная **flag** служит для того, чтобы определить, нашли мы нужный элемент или нет. Если нашли элемент, равный x , надо присвоить той переменной значение 1 и выйти из цикла. При этом в переменной m остается номер найденного элемента.

Если массив маленький, то скорость двоичного поиска незначительно отличается от линейного. Представим себе, что размер массива — 1000000 элементов и нужный нам элемент стоит в конце массива (это самый худший вариант для линейного поиска).

Размер массива, N	Число сравнений K	
	Линейный поиск	Двоичный поиск
10	≤ 10	≤ 4
1 000	$\leq 1\,000$	≤ 10
1 000 000	$\leq 1\,000\,000$	≤ 20
N	$\leq N$	$\leq \log_2 N$

Таким образом, чем больше элементов в массиве, тем выгоднее использовать двоичный поиск, поскольку число операций возрастает как логарифм N , медленнее, чем увеличивается размер массива. Его недостатком является то, что элементы должны быть заранее отсортированы. Двоичный поиск используется при поиске информации в больших базах данных.



Массивы в процедурах и функциях

Массивы, так же как и простые переменные, можно передавать в процедуры и функции в качестве параметров. Рассмотрим, например, функцию, вычисляющую среднее значение элементов массива. Желательно сделать ее так, чтобы в нее можно было передавать массивы любого размера и она всегда правильно вычисляла результат. Для этого функция должна знать

(или определить) размер массива. В языке Си функции не могут самостоятельно определять размер массива, поэтому он должен быть обязательно одним из параметров.

```
int Sum ( int A[], int N )
{
    int i, sum;

    sum = 0;
    for ( i = 0; i < N; i ++ )
        sum += A[i];

    return sum;
}
```

размер массива

массив-параметр

Обратите внимание, что в заголовке функции размер массива указан отдельно, нельзя объявлять массив-параметр как **A[N]**, а только как **A[]**. С другой стороны такая запись возможна только в заголовках функций, поскольку при этом не надо выделять новую память под массив. Объявлять локальный или глобальный массив, не указав явно его размер, нельзя.

Для вызова приведенной функции в параметрах надо указать название массива (без скобок) и его размер.

```
void main()
{
    int A[20], B[30], s;
    // здесь вводим массивы
    s := Sum(A, 20);
    printf("Сумма массива A %d, массива B %d", s, Sum(B, 30) );
}
```

вычислить сумму **A**

вычислить сумму **B**

2. Символьные строки



Что такое символьная строка?

Понятно, что символьная строка - это последовательность символов. В компьютерах используется 256 различных символов, каждый символ имеет свой код (от 0 до 255) по специальной таблице. Строка, как и другие переменные, записывается в память, причем компьютеру все равно, какие данные записаны – для него это набор байтов. Как же определить, где заканчивается строка? Есть два решения:

1. Хранить длину строки в отдельной ячейке (как в языке Паскаль).
2. Выбрать один особый символ, который будет обозначать конец строки, причем в середине строки этот символ не может встречаться.

В языке Си принят второй подход.

Символьная строка - это последовательность символом, которая заканчивается символом с кодом 0.

Символ с кодом ноль не имеет никакого изображения, в программе его записывают как `'\0'`.

Символ с кодом ноль (обозначается как `'\0'`) и цифра ноль (обозначается `'0'`, имеет код 48) это два разных символа



Объявление и инициализация

Строка представляет собой массив символов, поэтому и объявляется она именно так:

```
char s[80];
```

Однако строка отличается от массива тем, что она заканчивается символом с кодом 0 - признаком окончания строки, поэтому

Если массив символов будет использоваться как строка, надо выделять на 1 байт больше памяти

При выделении памяти глобальные переменные заполняются нулями, а локальные содержат "мусор". Начальное значение строки можно задать при объявлении в двойных кавычках после знака равенства:

```
char s[80] = "Привет, Вася!";
```

Символы в кавычках будут записаны в начало массива `s`, а затем - признак окончания строки `'\0'`. Оставшиеся символы не меняются и в локальных строках там будет мусор. Можно также написать так

```
char s[] = "Привет, Вася!";
```

В этом случае компилятор подсчитает символы в кавычках, выделит памяти на 1 байт больше и занесет в эту область саму строку и завершающий ноль. Аналогично можно выделить память на указатель:

```
char *s = "Привет, Вася!";
```

Результат - тот же самый, что и в предыдущем случае, но теперь `s` - это указатель и с ним можно работать так же, как с обычным указателем (присваивать, изменять и т.п.). Если строка не

будет изменяться во время работы программы, то можно объявить константу (постоянную строку) так:

```
#define PRIVET "Привет, Вася!"
```

и затем использовать имя **PRIVET** так, как переменную. В более современном языке Си++ можно то же самое сделать иначе:

```
const char PRIVET[] = "Привет, Вася!";
```

Стандартный ввод и вывод

Для ввода и вывода строк с помощью функций **scanf** и **printf** используется специальный формат "%s":

```
#include <stdio.h>
void main()
{
    char Name[50];
    printf("Как тебя зовут? ");
    scanf("%s", Name);
    printf("Привет, %s!", Name);
}
```

Заметьте, что в функцию **scanf** надо передать просто имя строки (без знака &), ведь имя массива является одновременно адресом его начального элемента.

Однако у функции **scanf** есть одна особенность: она заканчивает ввод, встретив первый пробел. Если вы на вопрос в предыдущем примере ввели "**Вася Печкин**", то увидите надпись "**Привет, Вася!**" вместо ожидаемого "**Привет, Вася Печкин!**". Если надо ввести всю строку целиком, включая пробелы (то есть до нажатия на клавишу **Enter**), придется делать иначе, заменив вызов **scanf** на

```
gets ( s );
```

Название этой функции происходит от английских слов *get string* - получить строку.

Для вывода строки на экран можно (кроме **printf**) использовать и функцию **puts**, которая после вывода строки еще и дает команду перехода на новую строку. В примере значение строки **Name** будет напечатано на следующей строчке экрана.

```
#include <stdio.h>
void main()
{
    char Name[50] = "Вася!";
    puts( "Привет, " );
    puts ( Name );
}
```

Задача. Ввести символьную строку и заменить в ней все буквы 'А' на буквы 'Б'.

Будем рассматривать строку как массив символов. Надо перебрать все элементы массива, пока мы не встретим символ с '\0' (признак окончания строки) и, если очередной символ - это буква 'А', заменить ее на 'Б'. Для этого используем цикл **while**, потому что мы заранее не знаем длину строки. Условие цикла можно сформулировать как «пока не конец строки».

```

#include <stdio.h>
void main()
{
    char a[80];
    int i;
    printf( "\n Введите строку \n" );
    gets ( s );
    i = 0;
    while ( s[i] != '\0' )
    {
        if ( s[i] == 'A' )
            s[i] = 'B';
    }
    puts ( "Результат:\n" );
    puts ( a );
}

```

начать с символа **s[0]**

пока не конец строки

меняем символ

Заметьте, что

Одиночный символ записывается в апострофах, а символьная строка – в кавычках.

При выводе строк с помощью функции `printf` часто применяется форматирование. После знака `%` в формате указывается размер поля для вывода строки. перед этим числом можно также поставить знак минус, что означает "прижать к левому краю поля".

Пример вывода	Результат	Комментарий
<code>printf ("[%s]", "Вася");</code>	[Вася]	На строку отводится минимально возможное число позиций.
<code>printf("[%6s]", "Вася");</code>	[Вася]	На строку отводится 6 позиций, выравнивание вправо.
<code>printf("[% -6s]", "Вася");</code>	[Вася]	На строку отводится 6 позиций, выравнивание влево.
<code>printf("[%2s]", "Вася");</code>	[Вася]	Строка не помещается в заданные 2 позиции, поэтому область вывода расширяется.



Работа с файлами

В реальной ситуации требуется обрабатывать очень много строк, которые чаще всего находятся в файле, причем их количество заранее неизвестно. Однако, если для обработки одной строки нам не требуется знать остальные, можно использовать способ, который мы применяли при работе с массивами данных неизвестного размера. В данном случае мы будем читать очередную строку из файла, обрабатывать ее и сразу записывать в выходной файл (если это требуется).

Работа с файлами имеет несколько особенностей. Во-первых, для чтения строки можно использовать функцию `fscanf`. Однако эта функция читает только одно слово и останавливается на первом пробеле. Поэтому

функция `fscanf` применяется тогда, когда надо читать файл по словам.

Вот пример чтения слова из открытого файла с указателем **fp**:

```
#include <stdio.h>
void main()
{
    char s[80];
    FILE *fp;
    fp = fopen ( "input.dat", "r" );
    fscanf ( fp, "%s", s );
    printf ( "Первое слово файла - %s", s );
    fclose ( fp );
}
```

Если надо читать всю строку с пробелами, используют функцию `fgets`. Она принимает три параметра:

- Имя символьной строки, в которую записать данные.
- Максимальную длину этой строки. Функция не допускает выхода за границы строки, если строка в файле длиннее, чем можно записать в память, читается только начальная часть, а остальное – при следующих вызовах `fgets`.
- Указатель на файл.

Если функция `fgets` не может прочитать строку из файла (например, если нет больше строк), то она возвращает в качестве результата специальное значение **NULL**. Это свойство можно использовать для обработки ошибок и прекращения ввода данных.

```
#include <stdio.h>
void main()
{
    char s[80];
    FILE *fp;
    fp = fopen ( "input.dat", "r" );
    if ( NULL == fgets ( s, 80, fp ) )
        printf ( "Не удалось прочитать строку" );
    else
        printf ( "Первое слово файла - %s", s );
    fclose ( fp );
}
```

Функция `fgets` читает строку из файла, пока не случится одно из двух событий:

- встретится символ перехода на новую строку '`\n`'
- прочитано столько символов, что они заняли всю строку (с учетом последнего нуля), например, в нашем случае она остановится, если прочтет 79 символов

В конце строки будет поставлен символ '`\0`'. Кроме того, если был найден символ перехода на новую строку '`\n`', он сохраняется и в строке `s`.

Задача. В каждой строке файла "input.dat" заменить все буквы 'А' на 'Б' и вывести измененный текст в файл "output.dat".

Обратите внимание, что в этой задаче файл может быть любой длины. Но мы можем обрабатывать строки последовательно одну за другой, так как для обработки одной строки не нужны предыдущие и следующие.

Для чтения из файла используем цикл `while`. Он заканчивает работу, если функция `fgets` вернет значение **NULL**, то есть все строки обработаны.

```

#include <stdio.h>
void main()
{
    char s[80];
    int i;
    FILE *fin, *fout;

    fin = fopen ( "input.dat", "r" );
    fout = fopen ( "output.dat", "w" );

    while ( NULL != fgets ( s, 80, fin ) )
    {
        i = 0;
        while ( s[i] != '\0' )
            if ( s[i] == 'A' ) s[i] = 'B';

        fprintf ( fout, "%s", s );
    }

    fclose ( fin );
    fclose ( fout );
}

```

← читаем строку

← обработаем строку

← выводим строку в файл

Обратите внимание, что мы не поставили символ перехода на новую строку при вызове функции `fprintf`. Это связано с тем, что при чтении функция `fgets` сохраняет символ `'\n'` в конце каждой строки (кроме последней), поэтому строки будут выведены в выходной файл так же, как они были записаны в исходном.



Функции для работы со строками

В языке Си есть достаточно много специальных функций, которые работают со строками - последовательностями символом с нулевым символом на конце. Для использования этих функций надо включить в программу заголовочный файл

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

Многие из этих функций достаточно опасны при неправильном использовании, ведь они не проверяют, достаточно ли выделено памяти для копирования, перемещения или другой операции, единственным признаком окончания строки для них является символ `'\0'`.



Длина строки - `strlen`

Это самая простая функция, которая определяет, сколько символов в переданной ей строке (не считая завершающего нуля). Ее имя происходит от английских слов *string length* (длина строки).

```

#include <stdio.h>
#include <string.h>
void main()
{
    int l;
    char s[] = "Prodigy";
    l = strlen(s);
    printf ( "Длина строки %s равна %d", s, l );
}

```

В этом примере функция определит, что длина строки равна 7. Теперь рассмотрим более сложную задачу.

Задача. В текстовом файле **"input.dat"** записаны строки текста. Вывести в файл **"output.dat"** в столбик длины этих строк.

```
#include <stdio.h>
#include <string.h>
void main()
{
    char s[80];
    FILE *fin, *fout;

    fin = fopen ( "input.dat", "r" );
    fout = fopen ( "output.dat", "w" );

    while ( NULL != fgets ( s, 80, fin ) )
    {
        fprintf ( fout, "%d\n", strlen(s) );
    }

    fclose ( fin );
    fclose ( fout );
}
```

← читаем строку из файла

← выводим ее длину в файл

Несмотря на то, что с первого взгляда программа написана верно, числа в файле будут на единицу больше, чем длины строк (кроме последней строки). Вспомнив предыдущий материал, объясните это. Далее будет показано, как получить точный результат.

Сравнение строк - **strcmp**

Для сравнения двух строк используют функцию **strcmp** (от английских слов **string comparison** - сравнение строк). Функция возвращает ноль, если строки равны (то есть "разность" между ними равна нулю) и ненулевое значение, если строки различны. Сравнение происходит по кодовой таблице **ASCII** (*American Standard Code for Information Interchange*), поэтому функция различает строчные и заглавные буквы - они имеют разные коды.

```
#include <stdio.h>
#include <string.h>
void main()
{
    char s1[] = "Вася",
        s2[] = "Петя";

    if ( 0 == strcmp(s1,s2) )
        printf("Строки %s и %s одинаковы", s1, s2);
    else printf("Строки %s и %s разные", s1, s2);
}
```

Если строки не равны, функция возвращает «разность» между первой и второй строкой, то есть *разность кодов первых различных символов*. Эти числа можно использовать для сортировки строк – если «разность» отрицательна, значит первая строка «меньше» второй, то есть стоит за ней в алфавитном порядке. В таблице показано несколько примеров (код буквы **'А'** равен **65**, код буквы **'В'** – **66**, код буквы **'С'** – **67**).

s1	s2	результат strcmp(s1, s2)
AA	AA	0
AB	AAB	'В' – 'А' = 66 – 65 = 1
AB	CAA	'А' – 'С' = 65 – 67 = - 2
AA	AAA	'\0' – 'А' = - 65

Задача. Ввести две строки и вывести их в алфавитном порядке.

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

void main()
{
    char s1[80], s2[80];

    printf ("Введите первую строку");
    gets(s1);

    printf ("Введите вторую строку");
    gets(s2);

    if ( strcmp(s1,s2) <= 0 )
        printf("%s\n%s", s1, s2);
    else printf("%s\n%s", s2, s1);
}
```

Иногда надо сравнить не всю строку, а только первые несколько символов. Для этого служит функция `strncmp` (с буквой *n* в середине). Третий параметр этой функции - количество сравниваемых символов. Принцип работы такой же - она возвращает нуль, если заданное количество первых символов обеих строк одинаково.

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

void main()
{
    char s1[80], s2[80];

    printf ("Введите первую строку");
    gets(s1);

    printf ("Введите вторую строку");
    gets(s2);

    if ( 0 == strncmp(s1, s2, 2) )
        printf("Первые два символа %s и %s одинаковы", s1, s2);
    else
        printf("Первые два символа %s и %s разные", s1, s2);
}
```

Один из примеров использования функции `strcmp` – проверка пароля. Составим программу, которая спрашивает пароль и, если пароль введен неверно, заканчивает работу, а если верно – выполняет какую-нибудь задачу.

Задача. Составить программу, которая определяет, сколько цифр в символьной строке. Программа должна работать только при вводе пароля «куку».

```

#include <stdio.h>
#include <string.h>
void main()
{
    char pass[] = "куку", s[80];
    int i, count = 0;

    printf ("Введите пароль ");
    gets(s);
    if ( strcmp ( pass, s ) != 0 )
    {
        printf ( "Неверный пароль" );
        return;
    }

    printf ("Введите строку");
    gets(s);

    i = 0;
    while ( s[i] != '\0' ) {
        if ( s[i] >= '0' && s[i] <= '9' )
            count ++;
    }

    printf("\nНашли %d цифр", count);
}

```

правильный пароль

вспомогательная строка

В этой программе использован тот факт, что коды цифр расположены в таблице символов последовательно от '0' до '9'. Поэтому можно использовать двойное неравенство, а не сравнивать текущий символ *s[i]* с каждой из цифр. Обратите внимание на разницу между символами '\0' (символ с кодом 0, признак конца строки) и '0' (символ с кодом 48, цифра 0). Переменная *count* работает как счетчик. Подумайте, как можно подсчитать, сколько в строке шестнадцатеричных цифр.

Копирование строк

Часто надо записать новое значение в строку или скопировать информацию из одной строки в другую. Функции копирования принадлежат к числу "опасных" - они могут вызвать сбой и зависание компьютера, если произойдет **выход за границы массива**. Это бывает в том случае, если строка, в которую копируется информация, имеет недостаточный **размер** (под нее выделено мало место в памяти).

В копировании участвуют две строки, они называются "*источник*" (строка, откуда копируется информация) и "*приемник*" (куда она записывается или добавляется).

При копировании строк надо проверить, чтобы для строки-приемника было выделено достаточно места в памяти

Простое копирование выполняет функция *strcpy*. Она принимает два аргумента: сначала строка-приемник, потом - источник (порядок важен!).

```

char s1[50], s2[10];
gets(s1);
strcpy ( s2, s1 );
puts ( s2 );

```

Источник (откуда)

Приемник (куда)

Этот фрагмент программы является "опасным" с точки зрения выхода за границы строки. В строку **s1** можно безопасно записать не более 49 символов (плюс завершающий ноль). Поэтому если с клавиатуры будет введена строка длиннее 49 символов, при записи ее в память произойдет выход за границы строки **s1**. Строка **s2** может принять не более 9 символов, поэтому при большем размере **s1** произойдет выход за границы строки **s2**.

Поскольку реально в функции передается адрес начала строки, можно заставить функцию начать работу любого символа от начала строки. Например, следующая строка скопирует строку **s2** в область памяти строки **s1**, которая начинается с ее 6-ого символа, оставив без изменения первые пять.

```
strcpy ( s1+5, s2 );
```

При этом надо следить, чтобы не выйти за границу массива. Кроме того, если до выполнения этой операции в строке **s1** меньше 5 символов, фокус не удастся.

Еще одна функция позволяет скопировать только заданное количество символов, она называется `strncpy` и принимает в третьем параметре количество символов, которые надо скопировать. Важно помнить, что эта функция НЕ записывает завершающий ноль, а только копирует символы (в отличие от нее `strcpy` всегда копирует завершающий ноль). Эта функция особенно полезна тогда, когда надо по частям собрать строку из кусочков.

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

```
void main()
```

```
{
    char s1[] = "Ку-ку", s2[10];
```

куда

откуда

сколько

```
    strncpy ( s2, s1, 2 );
```

```
    puts ( s2 );
```

```
    s2[2] = '\0';
```

```
    puts (s2);
```

```
}
```

Ошибка: в конце строки
нет символа `'\0'`



Проблемы при копировании строк

При копировании стандартные функции `strcpy` и `strncpy` поступают так: определяют количество символов, которые надо скопировать и затем переписывают их, начиная с первого символа до последнего. Если области источника и приемника не перекрываются, то все проходит нормально. Но попробуем "раздвинуть" строку, например для того, чтобы вставить что-то в ее середину. Пусть в строке **s1** записано имя и фамилия человека, а в строке **s2** - его отчество. Надо получить в строке **s1** полностью имя, фамилию и отчество.

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

void main()
{
    int n;
    char s1[80] = "Иван Рождественский",
          s2[] = "Петрович ";
    n = strlen(s2);
    strcpy (s1+5, s1+5+n);
    strncpy(s1+5, s2, n);
    puts ( s1 );
}
```

нашли длину второй строки

пытаемся раздвинуть на *n* символов

копируем отчество в середину

При первом вызове `strcpy` мы хотели скопировать конец строки, начиная с символа с номером 5 (он шестой с начала, так как нумерация идет с нуля) вправо на *n* символов (объявленная длина массива символов - 80 - позволяет это сделать). Однако из-за того, что копирование выполнялось с начала блока данных, скопировав на новое место первый символ фамилии ('Р') функция стерла букву 'н' (справа на 9 символов) и т.д. В результате получили

```
s1 = "Иван РождествеРождествеРождес"
```

В следующей строчке мы скопировали в середину отчество (без завершающего нуля) и получили

```
s1 = "Иван Петрович РождествеРождес"
```

Таким образом, вся задумка не удалась из-за того, что функция копирования работает в данном случае неверно. Выход из этой ситуации такой - написать свою функцию копирования, которая копирует не с начала блока, а с конца (однако она будет неверно работать в обратной ситуации - при сжатии строки). Например, так:

```
void strcpy1 ( char s1[], char s2[] )
{
    int n = strlen(s2);
    while ( n >= 0 )
    {
        s1[n] = s2[n];
        n --;
    }
}
```

Заметьте, что завершающий нуль строки *s2* также копируется. Если использовать в нашем примере эту функцию вместо *strcpy*, то получим желаемый результат.

Возникает вопрос: можно ли сделать функцию, которая всегда правильно копирует? Конечно, можно, хотя это и не просто. Для этого в самой функции надо использовать вспомогательную строку (ее называют *буфер*) и в ней формировать результат так, чтобы в процессе копирования не портилась исходная строка. Когда результат в буфере готов, его останется скопировать в то место, где была исходная строка и удалить память, выделенную под буфер. Попробуйте написать такую функцию, если известно, что ее будут использовать для копирования строк длиной не более 80 символов.



Объединение строк

Еще одна функция - `strcat` (от *string concatenation* - сцепка строк) позволяет добавить строку источник в конец строки-приемника (завершающий нуль записывается автоматически). Надо только помнить, что приемник должен иметь достаточный размер, чтобы вместить обе исходных строки.

Функция `strcat` автоматически добавляет в конец строки-результата завершающий символ `"\0"`.

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

void main()
{
    char s1[80] = "Могу, ",
          s2[] = "хочу, ", s3[] = "надо!";

    strcat ( s1, s2 );
    puts ( s1 );
    strcat ( s1, s3 );
    puts ( s1 );
}
```

куда откуда

результат
s1 = "Могу, хочу "

результат
s1 = "Могу, хочу, надо!"

Заметьте, что если бы строка **s1** была объявлена как **s1[]** (или с длиной меньше 18), произошел бы выход за границы массива с печальными последствиями.

Задача. Ввести с клавиатуры имя файла. Изменить его расширение на **".exe"**.

Алгоритм решения:

1. Найти в имени файла точку `'.'` или признак конца строки `'\0'`.
2. Если нашли точку, скопировать начиная с этого места новое расширение **".exe"** (используем функцию `strcpy`).
3. Если нашли конец строки (точки нет), добавить в конец расширение **".exe"** (используем функцию `strcat`).

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

void main()
{
    char s[80];
    int n;

    printf("Введите имя файла ");
    gets ( s );

    n = 0;
    while ( (s[n] != '.') && (s[n] != '\0') )
        n++;

    if ( s[n] == '.' )
        strcpy ( s + n, ".exe" );
    else strcat ( s, ".exe" );

    puts ( s );
}
```

номер символа `'.'`

ищем первую точку

до конца строки

нашли точку

Задача. Ввести с клавиатуры фамилию и имя в одну строку (допустим, **Иванов Вася**). Построить в памяти строку «**Привет, Вася Иванов!**».

Алгоритм решения этой задачи можно записать так:

1. Ввести строку с клавиатуры (назовем ее ***a***).

S	И	В	а	Н	о	В		В	а	с	я	\0							
----------	---	---	---	---	---	---	--	---	---	---	---	----	--	--	--	--	--	--	--

2. Найти длину первого слова (фамилии), обозначим ее через n . Тогда символ с номером n – это пробел между именем и фамилией.

Diagram illustrating the insertion of a new character 'n' into a string 's' at index 6. The string 's' is represented as an array of characters: 'И', 'в', 'а', 'н', 'о', 'в', followed by a gap, then 'Б', 'а', 'с', 'я', and '\0'. An arrow points from a box containing 'n' to the gap in the string.

3. Запишем в новую строку первую часть фразы – «Привет, », используя функцию `strcpy`.

[illegible]

4. Добавим в конец этой строки второе слово (имя), используя функцию `strcat`. Для этого надо скопировать в конец строки `a` все символы строки `s`, начиная с `n+1`-ого до конца строки.

[illegible]

5. Добавим в конец строки пробел, используя функцию `strcat`.

а	П	р	и	в	е	т	,		В	а	с	я	№							
---	---	---	---	---	---	---	---	--	---	---	---	---	---	--	--	--	--	--	--	--

6. Определим длину полученной строки и обозначим ее через **len**. Символ с номером **len** – это символ с кодом '\0'.

а

П	р	и	в	е	т	,		В	а	с	я		\0						
---	---	---	---	---	---	---	--	---	---	---	---	--	----	--	--	--	--	--	--

len

↓

7. Добавим в конец строки первое слово (фамилию), используя функцию `strncpy`. Для этого скопируем в конец строки **a** первые **n** символов строки **s**. Обратите внимание, что в конце строки теперь нет завершающего символа `'\0'`.

а

П	р	и	в	е	т	,		В	а	с	я		И	в	а	н	о	в	?	?
---	---	---	---	---	---	---	--	---	---	---	---	--	---	---	---	---	---	---	---	---

$len+n$

8. Осталось добавить в конец строки символ `'!'` и `'\0'`. Для этого используется функция **`strcpy`**. Для копирования нужно использовать адрес **`s+len+n`**, так как к строке длиной **`len`** мы добавили фамилию длиной **`n`**.

П	р	и	в	е	т	,		В	а	с	я		И	в	а	н	о	в	!	\0
---	---	---	---	---	---	---	--	---	---	---	---	--	---	---	---	---	---	---	---	----

В программе используется важное свойство массивов в языке Си (в том числе и символьных строк, которые являются массивами символов):

Если массив называется s , то запись $s+i$ обозначает адрес элемента $s[i]$.

Так как функциям `strcpy` и `strcat` надо передать адреса в памяти, куда и откуда переместить данные, мы можем использовать запись **`a+len`** вместо **`&a[len]`** и т.д.

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

```
void main()
{
```

```
    char s[80], a[80] = "Привет, ";
    int n, len;
```

```
    printf("Введите фамилию и имя ");
    gets ( s );
```

```
    n = 0;
    while ( (s[n] != ' ') && (s[n] != '\0') )
        n ++;
```

```
    if ( s[n] != ' ' )
    {
        printf( "Неверная строка" );
        return;
    }
```

```
    strcat ( a, s + n + 1 );
```

```
    strcat ( a, " ");
```

```
    len = strlen ( a );
```

```
    strncpy ( a + len, s, n );
```

```
    strcpy ( a + len + n, "!" );
```

```
    puts ( a );
}
```

начало строки

ищем первый пробел

до конца строки

нет пробела - ошибка

добавили имя

добавили пробел

нашли длину строки

добавили фамилию

добавили "!"



Поиск в строках

Когда говорят о поиске в строках, обычно рассматривают две задачи: найти первый заданный символ с начала (или с конца), или также найти заданную подстроку (если она есть). Первую задачу выполняют функции `strchr` (поиск с начала строки) и `strrchr` (поиск с конца строки), а вторую - функция `strstr`.

Все эти функции возвращают указатель на найденный символ (или на первый символ найденной подстроки). Это значит, что переменная, в которую записывается это значение, должна быть объявлена как *указатель* на символьную переменную. Мы уже встречались с указателями, когда работали с файлами. Указатель – это ячейка в памяти, в которую можно записывать *адрес* другой переменной.

Структура вызова функций такая: на первом месте - где искать (строка), на втором - что искать (один символ для функций `strchr` и `strrchr` или строка для `strstr`). Чтобы получить номер символа с начала строки, надо вычесть из полученного указателя адрес начала массива. Если поиск завершился неудачно, функции возвращают **NULL**.


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

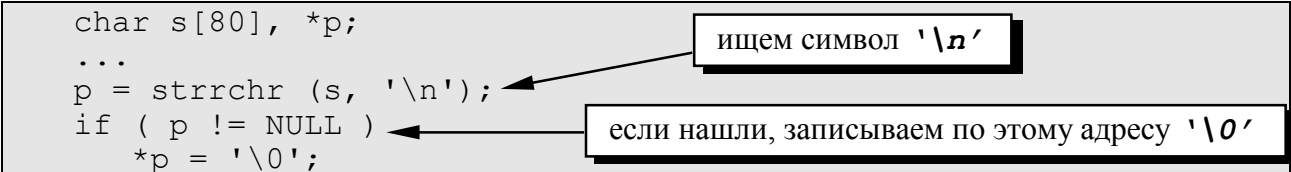
void main()
{
    char s1[] = "Мама мыла раму",
          s2[] = "Война и мир", *p;

    p = strchr(s1, 'a');
    if ( p != NULL ) {
        printf("Первая буква а: номер %d", p - s1);
        p = strrchr(s1, 'a');
        printf("\nПоследняя буква а: номер %d", p - s1);
    }

    p = strstr( s2, "мир");
    if ( p != NULL )
        printf("\nНашли мир в %s", s2);
    else
        printf("\nНет слова мир в %s", s2);
}
```

Вспомните, что при чтении строк из файла с помощью функции `fgets` на конце иногда остается символ перехода на новую строку `'\n'`. Чаще всего он совсем не нужен и надо его удалить - поставить на его месте ноль. Делается это так:

```
char s[80], *p;
...
p = strrchr (s, '\n');
if ( p != NULL )
    *p = '\0';
```



Рассмотрим теперь более сложную задачу.

Задача. С клавиатуры вводится предложение и слово. Надо определить, сколько раз встречается это слово в предложении.

Функция `strstr` может определить только первое вхождение слова в строку, поэтому в одну строчку эту задачу не решить.

Попробуем использовать такую идею: если мы нашли адрес первого данного слова в строке и записали его в указатель ***p***, то искать следующее слово имеет смысл только начиная с адреса ***p + длина слова***. Повторяем эту операцию в цикле, пока функция `strstr` может найти слово в оставшейся части строки. Поскольку начало области поиска постоянно смещается с каждым новым найденным словом, адрес оставшейся части надо хранить в отдельной переменной типа "указатель на символ". Реализация может выглядеть так:

```
#include <stdio.h>
#include <string.h>
void main()
{
    int len, count;
    char s[80], word[20],
          *p,
          *start;

    puts ( "Введите предложение" );
    gets ( s );
```

```
puts ( "Введите слово для поиска" );
gets ( word );
```

```
len = strlen ( word );
```

нашли длину слова

```
count = 0;
start = s;
```

ищем с начала строки

```
while ( 1 ) {
    p = strstr ( start, word );
    if ( p == NULL ) break;
    count ++;
    start = p + len;
}
```

есть ли еще слово?

таких слов больше нет, выход

отсюда ищем следующее слово

```
printf ( "В этом предложении %d слов %s", count, word );
}
```

В конце работы цикла в переменной **count**, будет записано количество данных слов в предложении. Заметьте, что вместо переменной **start** можно везде использовать **p**, результат от этого не изменится.

Форматирование строк

В программах часто требуется перед выводом информации сформировать всю строку для вывода целиком, включив в нее все необходимые данные. Например, сообщение об ошибке выводится стандартной функцией, и в это сообщение надо включить числовые данные. Другой пример - вывод текста в графическом режиме, для которого нет аналога функции `printf`.

В этих случаях необходимо использовать функцию `sprintf`, которая поддерживает те же форматы данных, что и `printf`, но записывает результат не на экран и не в файл, а в *символьную строку* (под нее надо заранее выделить память). Вот как выглядит вывод на экран значения переменных **x** и **y** в графическом режиме:

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

```
void main()
```

выделить память под строку

```
{
    char s[80];
    int x, y;
```

```
    // включить графический режим
```

```
    x = 1;
    y = 5;
```

здесь записать результат

список переменных

```
sprintf (s, "X=%d, Y=%d", x, y);
```

```
outtextxy ( 100, 100, s );
```

```
getch();
closegraph();
}
```

Не забудьте, что для использования функции `outtextxy` надо включить монитор в графический режим.

Чтение из строки

Иногда, особенно при чтении данных из файлов, возникает обратная задача: есть символьная строка, в которой записаны данные. Необходимо ввести их в соответствующие ячейки памяти.

В этом случае используется функция `sscanf`, которая читает данные по указанному формату не с клавиатуры (как `scanf`) и не из файла (как `fscanf`), а из символьной строки. В приведенном ниже примере мы ищем в файле строчку, которая начинается с символа '#' и считываем из нее значения **x** и **y**.

Сложность задачи заключается в том, что мы не знаем точно, какая по счету эта строчка в файле. Если не использовать функцию `sscanf`, то пришлось бы сначала найти номер нужной строки в файле, затем начать просмотр с начала, отсчитать нужное количество строк и использовать `fscanf`.

```
#include <stdio.h>
void main()
{
    char s[80];
    int x, y;
    FILE *fp;

    fp = fopen ( "input.dat", "r" );
    while ( fgets ( s, 80, fp ) )
        if ( s[0] == '#' ) {
            sscanf ( s+1, "%d%d", &x, &y);
            break;
        }

    fclose ( fp );
    printf ( "x = %d, y = %d", x, y );
}
```

выделить память под строку

нашли '#' -
прочитать данные

пропустить '#'

Строки в функциях и процедурах

Стандартные и ваши собственные функции могут принимать строки в качестве параметров. Здесь, однако, есть некоторые тонкости.

Вы знаете, что если параметр функции объявлен как **int a**, то изменение переменной **a** в функции никак не влияет на ее значение в вызывающей программе - функция создает копию переменной и работает с ней (для того, чтобы функция могла это сделать, надо объявить параметр как **int &a**). Это относится ко всем простым типам.

Когда вы передаете в функцию или процедуру строку, вы в самом деле передаете *адрес начала строки*, никаких копий строки не создается. Поэтому всегда надо помнить, что

При изменении строки-параметра в функции или процедуре меняется и соответствующая строка в основной программе


Как же объявить строку-параметр в своей функции? В простейшем варианте - так же, как и массив символов: `char s[]`. А можно также и как указатель (все равно в функцию передается адрес строки `char *s`). Между этими двумя способами есть некоторая разница. В первом случае `s` - это имя массива символов, поэтому нельзя его изменять, а во втором случае - указатель на символ, который можно сдвигать, как хочется.

Все стандартные функции языка Си объявляют символьные строки как указатели - это дает большую свободу действий. Сравните, например, две реализации функции, которая

копирует строки (аналогично `strcpy`). Первая выполнена с помощью параметра-массива, а вторая - с помощью указателя.

```
void copy1 ( char s1[],
             char s2[] )
{
    int i = 0;
    while ( s2[i] ) {
        s1[i] = s2[i];
        i ++;
    }
    s1[i] = '\0';
}
```

```
void copy2 ( char *s1, char *s2 )
{
    while ( *s1++ = *s2++ );
}
```

- 
- 1) скопировать **s2* по адресу *s1*
 - 2) увеличить *s1* и *s2*
 - 3) делать так пока **s2* не ноль

Как видите, вторая функция получилась более компактной. Применение указателей позволило не вводить дополнительную переменную, хотя и сделала программу менее ясной. Итак, в условии цикла `while` стоит оператор присваивания. Если не обращать внимания на плюсы, он означает "взять символ по адресу *s2* и записать его по адресу *s1*". Двойные плюсы ПОСЛЕ *s1* и *s2* означают, что ПОСЛЕ выполнения присваивания оба **указателя** надо увеличить на единицу, то есть перейти к следующему символу.

Что же является условием цикла? Оказывается условие - это величина **s1*, то есть код символа по адресу *s1*. Когда же происходит проверка? Это зависит от расположения знаков *++*. В данном случае они стоят ПОСЛЕ имен переменных, поэтому операция инкремента выполняется ПОСЛЕ проверки условия. Проверка выполняется так: скопировали очередной символ, посмотрели на него, и если он - ноль (признак конца строки), то вышли из цикла. После этого увеличили указатели *s1* и *s2*. Обратите внимание, что после выхода из цикла увеличение указателей также происходит, и они будут указывать не на нули, завершающие строки, а на следующие байты в памяти.

3. Матрицы (двухмерные массивы)



Что такое матрица?

Вспомните, что из себя представляет адрес любого человека (или фирмы). Вы можете сказать: "Я живу на Невском проспекте в доме 20 в квартире 45", но никому в голову не придет сказать: "Я живу в 13678 квартире от начала Невского проспекта". Почему? Потому что неудобно. Первая часть адреса - улица, вторая - дом, третья - квартира.

Часто обычный массив неудобен для хранения данных из-за своей линейной структуры. Например, пусть нам надо обрабатывать информацию о количестве выпавших осадков за несколько лет, причем известны осадки по месяцам. Если обрабатывать данные вручную, то удобнее всего использовать таблицу - по горизонтали откладывать года, а по вертикали - месяцы (или наоборот). В принципе в программе можно использовать и одномерный массив, но тогда требуется пересчитывать индексы - по известному году и месяцу получать смещение ячейки от начала линейного массива:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
янв	фев	мар	апр	май	июн	июл	авг	сен	окт	ноя	дек	янв	фев	мар	апр	май	июн	ию
2000												2001						

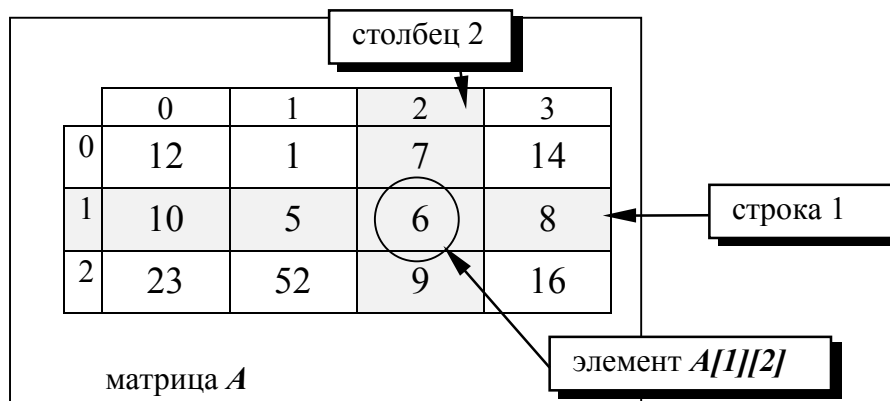
Для такого пересчета можно было бы использовать формулу

$$i = (\text{год} - 2000) * 12 + \text{месяц} - 1;$$

где i - индекс нужного элемента массива. Тем не менее так не делают (почти никогда), потому что для работы с табличными данными во всех современных языках программирования существуют *двухмерные массивы* или *матрицы*.

Матрица - это прямоугольная таблица чисел. В информатике матрица представляется в виде двухмерного массива, то есть массива, все элементы которого имеют два индекса.

Матрица, как и таблица, состоит из строк и столбцов. Два индекса элемента - это и есть номера строки и столбца, на пересечении которых этот элемент находится.



В языке Си каждый индекс записывается отдельно в квадратных скобках. Каждую строку и каждый столбец матрицы можно рассматривать как обычный одномерный массив. Поэтому можно сказать, что матрица - это *массив из массивов*. Существует только два ограничения:

1. Все элементы матрицы должны быть **одинакового типа**.
2. **Длина всех строк** матрицы должна быть **одинакова** (также как и высота всех столбцов).

Первый индекс элемента матрицы – это *строка*, второй – *столбец*. Поэтому когда говорят о "матрице 4 на 5", это означает, что матрица имеет 4 строки и 5 столбца.

Матрицы, у которых число строк равно числу столбцов, называют *квадратными*. В квадратных матрицах можно выделить *главную диагональ* - это все элементы, у которых номер строки равен номеру столбца, то есть

$$A[0][0], A[1][1], \dots, A[N-1][N-1]$$

для матрицы размером N на N .



Объявление матриц

Матрицы объявляются также, как и простые массивы, но у их не один индекс, а два. При объявлении в отдельных квадратных скобках указывается количество строк и количество столбцов. Например, оператор

```
int Bmv[20][10];
```

выделит место в памяти под матрицу целых чисел, имеющую 20 строк и 10 столбцов (всего $20 \times 10 = 200$ элементов). Если матрица глобальная (объявляется выше всех процедур и функций), то она в самом начале заполняется нулями. Локальные матрицы (объявленные внутри процедуры или функции) содержат "мусор" – неизвестные значения.



Начальные значения элементов

При объявлении можно сразу задать все или часть ее элементов, например так

```
float X[2][3] = {{1., 2., 3.}, {4., 5., 6.}};
```

Как видно из примера, элементы каждой строки заключаются в отдельные фигурные скобки. Если задать не все элементы, то остальные заполнятся нулями:

```
float X[2][3] = {{1., 3.}, {6.}};
```

Здесь элементы $X[1][2]$, $X[2][1]$, и $X[2][2]$ будут нулевыми.



Расположение матриц в памяти

Иногда бывает полезно знать, как матрицы располагаются в памяти ЭВМ. Оказывается во всех современных языках программирования (кроме *Фортрана*) элементы матрицы располагаются *по строкам*, то есть сначала изменяется последний индекс. Объявленная выше матрица X расположена так:

$X[0][0]$	$X[0][1]$	$X[0][2]$	$X[1][0]$	$X[1][1]$	$X[1][2]$
-----------	-----------	-----------	-----------	-----------	-----------



Стандартный ввод и вывод

Как и для одномерных массивов, матрицы могут быть введены с клавиатуры, из файла, и заполнены с помощью случайных чисел. Общий принцип - для каждого элемента функция чтения вызывается отдельно. Представим себе матрицу как массив строк равной длины. Для ввода одной строки требуется цикл, и таких строк несколько. Поэтому для работы с матрицами требуется *двойной* или *вложенный* цикл, то есть цикл в цикле.

Ввод с клавиатуры

Единственная проблема состоит в том чтобы не перепутать переменные в двух циклах и пределы их изменения.

```
#include <stdio.h>

const M = 5; // число строк
const N = 4; // число столбцов

void main()
{
    int i, j, A[M][N];

    for ( i = 0; i < M; i ++ )
        for ( j = 0; j < N; j ++ ) {
            printf ("A[%d][%d]= ", i, j);
            scanf ("%d", & A[i][j]);
        }
    ... // работа с матрицей
}
```

цикл по строкам

цикл по столбцам

подсказка

ВВОД **A[i][j]**

Заметьте, что при изменении порядка циклов (если поменять местами два оператора `for`) изменится и порядок ввода элементов в память.

Заполнение случайными числами

Выполняется также в двойном цикле аналогично одномерным массивам. В примере показано заполнение целой матрицы случайными числами в интервале $[a, b]$ (для вещественных чисел формула изменится - см. одномерные массивы).

В этой и следующей программах мы будем считать, что объявлена целая матрица **M** на **N**, где **M** и **N** — целые константы (объявленные через **#define** или **const**), а также целые переменные **i** и **j**.

```
for ( i = 0; i < M; i ++ )
    for ( j = 0; j < N; j ++ )
        A[i][j] = random(b-a+1) + a;
```

Вывод на экран

При выводе матрицы ее элементы желательно расположить в привычном виде — по строкам. Напрашивается такой прием: вывели одну строку матрицы, перешли на новую строку экрана, и т.д. Надо учитывать, что для красивого вывода на каждый элемент матрицы надо отвести равное количество символов (иначе столбцы будут неровные). Делается это с помощью форматирования — цифра после знака процента задает количество символов, отводимое на данное число.

```
printf ("Матрица A\n");

for ( i = 0; i < M; i ++ ) {
    for ( j = 0; j < N; j ++ )
        printf ( "%4d", A[i][j] );
    printf ("\n");
}
```

вывод строки

4 позиции экрана на
каждое число

на новую строку



Работа с файлами



Текстовые файлы

При вводе из текстового файла надо читать последовательно все элементы, обрабатывая (так же, как и для линейных массивов) ошибки отсутствия или недостатка данных в файле.

```
#include <stdio.h>

const M = 5; // число строк
const N = 4; // число столбцов

void main()
{
    int i, j, A[M][N];
    FILE *fp;

    fp = fopen("input.dat", "r");

    for ( i = 0; i < M; i ++ )
        for ( j = 0; j < N; j ++ )
            if ( 0 == fscanf (fp, "%d", & A[i][j]) )
            {
                puts("Не хватает данных");
                fclose ( fp );
                return;
            }

    fclose ( fp );
    ... // работа с матрицей
}
```

цикл по строкам

цикл по столбцам

ввод **A[i][j]**
fscanf вернет 0,
число не прочитано

закрывать файл при ошибке

закрывать файл нормально

Вывести матрицу в текстовый файл можно так же, как и на экран, только надо сначала открыть текстовый файл на запись, затем в двойном цикле использовать функцию `fprintf` вместо `printf`, и в конце закрыть файл (см. вывод одномерных массивов в файл).



Двоичные файлы

С двоичным файлом удобно работать тогда, когда данные записала (или будет читать) другая программа и их не надо просматривать вручную. Основное преимущество этого способа — скорость чтения и записи, поскольку весь массив читается (или записывается) сразу единым блоком. При этом функциям `fread` и `fwrite` надо указать размер одного элемента массива и количество таких элементов, то есть **M*N**.

В программе, которая приведена ниже, матрица читается из двоичного файла, затем с ней выполняются некоторые действия (они обозначены многоточием) и эта же матрица записывается в выходной файл.


```

#include <stdio.h>

const M = 5; // число строк
const N = 4; // число столбцов

void main()
{
    int total, A[M][N];
    FILE *fp;

    fp = fopen("input.dat", "rb");
    total = fread(A, sizeof(integer), M*N, fp);
    fclose ( fp );

    if ( total != M*N )
    {
        printf("Не хватает данных");
        return;
    }
    ... // работа с матрицей

    fp = fopen("output.dat", "wb");
    if ( M*N != fwrite(A, sizeof(integer), M*N, fp) )
        printf("Ошибка записи в файл");
    fclose ( fp );
}

```

чтение всего массива

обработка ошибки

запись всего массива

Для обработки ошибок используется тот факт, что функции `fread` и `fwrite` возвращают количество реально прочитанных (записанных) элементов, и если оно не равно заданному ($M*N$), то произошла ошибка.

Алгоритмы для работы с матрицами

Перебор элементов матрицы

В отличие от одномерных массивов, для перебора всех элементов матрицы надо использовать двойной цикл. Ниже показано, как найти минимальный элемент в массиве и его индексы. Сначала считаем, что минимальным является элемент $A[0][0]$ (хотя можно начать и с любого другого), а затем проходим все элементы, проверяя, нет ли где еще меньшего. Так же, как и для одномерного массива, запоминаются только индексы, а значение минимального элемента "вытаскивается" прямо из массива.

```

float A[M][N], i, j, row, col;
...

row = col = 0;
for ( i = 0; i < M; i ++ )
    for ( j = 0; j < N; j ++ )
        if ( A[i][j] < A[row][col] ) {
            row = i;
            col = j;
        }

printf ( "Минимальный элемент A[%d][%d]=%d",
        row, col, A[row][col] );

```

сначала считаем $A[0][0]$ минимальным

запомнили новые индексы

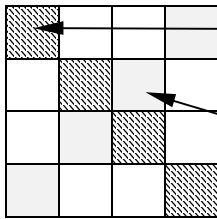


Работа с отдельными элементами

Рассмотрим квадратную матрицу N на N . Выведем на экран обе ее диагонали (главную диагональ и перпендикулярную ей). С главной диагональю все просто - в цикле выводим все элементы, у которых номера строки и столбца равны, то есть $A[i][i]$ для всех i от 0 до $N-1$. Вторую диагональ формируют такие элементы:

$$A[0][N-1], A[1][N-2], A[2][N-3], \dots, A[N-1][0]$$

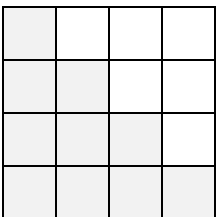
Обратим внимание, что каждый следующий элемент имеет номер строки на 1 больше, а номер столбца - на 1 меньше. Таким образом, **сумма номеров строки и столбца - постоянна** и равна $N-1$. Тогда, зная номер строки i можно сразу сказать, что на второй диагонали стоит ее элемент $A[i][N-1-i]$.



```
printf("Главная диагональ:\n");
for ( i = 0; i < N; i ++ )
    printf ("%d ", A[i][i]);

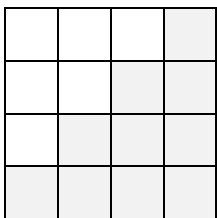
printf("\nВторая диагональ:\n");
for ( i = 0; i < N; i ++ )
    printf ("%d ", A[i][N-1-i]);
```

Теперь можно рассмотреть более сложный случай. Обнулим все элементы, кроме тех, которые стоят выше главной диагонали. В первой строке это единственный элемент $A[0][0]$, во второй - два элемента: $A[1][0]$ и $A[1][1]$, в третьей - три: $A[2][0]$, $A[2][1]$ и $A[2][2]$. Заметим, что в строке i обнуляются все элементы, у которых номера столбцов от 0 до i .



```
for ( i = 0; i < N; i ++ )
    for ( j = 0; j < i; j ++ )
        A[i][j] = 0;
```

Еще более сложно обработать все элементы, стоящие не выше второй диагонали. В первой строке это единственный элемент $A[0][N-1]$, во второй - два элемента: $A[1][N-2]$ и $A[1][N-1]$, в третьей - три: $A[2][N-3]$, $A[2][N-2]$ и $A[2][N-1]$. Заметим, что в строке i обнуляются все элементы, у которых номера столбцов от $N-1-i$ до $N-1$.



```
for ( i = 0; i < N; i ++ )
    for ( j = N-1-i; j < N; j ++ )
        A[i][j] = 0;
```



Перестановка строк и столбцов

Пусть надо переставить две строки с индексами $i1$ и $i2$. Это значит, что для каждого столбца j надо поменять местами элементы $A[i1][j]$ и $A[i2][j]$ через временную переменную (она называется **temp**).

```
for ( j = 0; j < N; j ++ ) {
    temp = A[i1][j];
    A[i1][j] = A[i2][j];
    A[i2][j] = temp;
}
```



Преобразование в одномерный массив

Иногда надо скопировать матрицу **A** размером **M** на **N** в одномерный массив **B** размером **M*N**. Очевидно, что при копировании по строкам (сначала первая строка, затем вторая и т.д.) элемент первой строки **A[0][j]** надо скопировать в **B[j]**, элементы второй строки **A[1][j]** - в **B[N+j]** и т.д. Отсюда следует, что для любой строки **i** элемент **A[i][j]** копируется в **B[i*N+j]**. теперь осталось только в двойном цикле перебрать все элементы матрицы.

```
for ( i = 0; i < M; i ++ )  
    for ( j = 0; j < N; j ++ )  
        B[i*N+j] = A[i][j];
```

Заметим, что если надо провести какую-то операцию со всеми или некоторыми (стоящими подряд) элементами матрицы и для одномерного массива уже есть соответствующая процедура, ее можно использовать, учитывая, что имя строки массива (например, **A[0]**) является указателем на начальный элемент этой строки. При этом надо учитывать, что в памяти элементы матрицы расположены по строкам. Например, функция вычисления суммы элементов (см. массивы) может применяться для матрицы так:

```
s0 = Sum(A[0], N);  
s14 = Sum(A[1], 4*N);  
sAll = Sum(A[0], M*N);
```

сумма строки 0

сумма строк 1-4

сумма всех элементов

4. Массивы символьных строк

Если строка - это массив символов, то массив строк – это массив из массивов или *двухмерный массив символов* (матрица). Как и в матрицах, длина всех строк (выделенная память) должна быть одинакова. Отличие массива строк от матрицы целых или вещественных заключается в том, что строка рассматривается не просто как набор символов, а как единое целое – символьная строка.



Объявление и инициализация

Ниже показано два способа объявления массива из 4 строк, в каждой из которых может быть до 80 символов (считая завершающие нули). Первый способ - это стандартное объявление двухмерного массива символов.

```
char s[4][80];
```

Во втором сначала определяется новый тип данных `str80` - строка длиной 80 символов, а потом объявляется массив из этих строк.

```
typedef str80[80];  
...  
str80 s[4];
```

новый тип - строка
длиной 80 символов

Напомним, что директива `typedef` должна стоять до первого использования нового типа. Принято ставить ее вне всех процедур и функций, там же, где объявляются глобальные переменные. Дальше каждую строку можно использовать отдельно. В том числе можно "добраться" до любого символа, вспомнив, что мы имеем дело с двухмерным массивом и `s[2][15]` – это 16-ый символ 3-ей строки (нумерация начинается с нуля!).

Всем или нескольким первым строкам массива можно задать начальные значения после знака равенства в фигурных скобках через запятую. Если начальных значений меньше, чем строк в массиве, оставшиеся последние строки заполняются нулями.

```
char s[4][80] = { "Вася", "Петя" };
```



Ввод и вывод

Для каждой строки ввод и вывод делается отдельно, то есть для ввода и вывода массива строк надо использовать цикл. Следующий фрагмент позволяет ввести не более 20 строк (пока не введена пустая строка). Переменная ***count*** будет хранить количество введенных строк.

```
char s[20][80];  
int i, count = 0;  
printf("Введите текст (Enter-все)\n");  
for ( i = 0; i < 20; i ++ ) {  
    gets (s[count]);  
    if ( s[count][0] == '\0' ) break;  
    count ++;  
}
```

проверить начальный
символ введенной строки

Вывод массива строк на экран, а также работа с текстовым файлом (ввод и вывод) выполняется аналогично в цикле.



Сортировка

Мы уже научились сортировать массивы целых и вещественных чисел, а теперь надо сортировать строки. Можно придумать разные способы, но наиболее часто встречается алфавитная сортировка. При сортировке строк возникают две проблемы:

1. Определить, какая из двух строк "меньше", то есть какая из двух должна стоять выше.
2. Сортировка массивов чисел предусматривает перестановку элементов массива. В случае строк это связано с копированием больших объемов данных, что крайне невыгодно.

Начнем с первой. Мы говорили, что есть функция сравнения строк `strcmp`, которая возвращает нуль, если строки равны, и не нуль, если они разные. Оказывается, эта функция возвращает "разность" этих строк, то есть *разность кодов* их первых отличающихся символов. Если функция вернула отрицательное число - первая строка "меньше" и стоит по алфавиту раньше, если положительное - наоборот. Здесь надо учитывать, что сравнение идет по таблице кодов, и коды заглавных букв меньше, чем коды строчных (и для русских, и для английских).

Вторая проблема более серьезная и решается с помощью *указателей*. Сначала выделим в памяти массив указателей на строки и сделаем так, чтобы *i*-ый указатель указывал на *i*-ую строку массива. Теперь достаточно правильно расставить указатели и сортировка будет выполнена – к строкам можно будет обращаться в алфавитном (или каком-либо другом) порядке с помощью этих указателей. Процедура сортировки методом пузырька выглядит так:

```
void SortStrings ( char *s[], int n )
{
    char *p;
    int i, j;
    for ( i = 0; i < n-1; i ++ )
        for ( j = n-1; j > i; j -- )
            if ( strcmp(s[j-1], s[j]) > 0 ) {
                p = s[j]; s[j] = s[j-1];
                s[j-1] = p;
            }
}
```

Diagram annotations for the SortStrings function:

- Arrow from `char *s[]` to "массив указателей на строки" (array of pointers to strings).
- Arrow from `int n` to "сколько строк" (how many strings).
- Arrow from the swap block (`p = s[j]; s[j] = s[j-1]; s[j-1] = p;`) to "перестановка указателей" (swap pointers).

Эту функцию использует приведенная ниже программа, которая сортирует строки выводит их на экран уже в алфавитном порядке.

```
void main()
{
    char s[20][80];
    char *ps[20];
    int i, count;
    ... // ввод строк, count - их количество

    for ( i = 0; i < count; i ++ )
        ps[i] = s[i];
    SortStrings ( ps, count );
    for ( i = 0; i < count; i ++ )
        puts ( ps[i] );
}
```

Diagram annotations for the main function:

- Arrow from `char s[20][80];` to "массив на 20 строк" (array of 20 strings).
- Arrow from `char *ps[20];` to "массив из 20 указателей на строки" (array of 20 pointers to strings).
- Arrow from `ps[i] = s[i];` to "расставили указатели" (set pointers).
- Arrow from `SortStrings (ps, count);` to "сортировка указателей" (sort pointers).
- Arrow from `puts (ps[i]);` to "вывод через указатели" (output via pointers).

5. Управление памятью



Указатели

Рассмотрим такую задачу: в файле **"data.dat"** записаны целые числа. Надо отсортировать их и записать в другой файл. Проблема заключается в том, что заранее неизвестно, сколько в файле таких чисел. В такой ситуации есть два варианта:

1. Выделить память с запасом, если известно, например, что этих чисел гарантированно не более 1000.
2. Использовать динамическое выделение памяти - сначала определить, сколько чисел в массиве, а потом выделить ровно столько памяти, сколько нужно.

Наиболее грамотным решением является второй вариант (использование динамических массивов). Для этого надо сначала поближе познакомиться с указателями.

Указатель - это переменная, в которой хранится адрес другой переменной или участка памяти.

Указатели являются одним из основных понятий языка Си. В такие переменные можно записывать адреса любых участков памяти, на чаще всего - адрес начального элемента динамического массива. Что же можно делать с указателями?

Объявить	<pre>char *pC; int *pI; float *pF;</pre>	Указатели объявляются в списке переменных, но перед их именем ставится знак *. Указатель всегда указывает на переменную того типа, для которого он был объявлен. Например, pC может указывать на любой символ, pI - на любое целое число, а pF - на любое вещественное число.
Присвоить адрес	<pre>int i, *pI; ... pI = &i;</pre>	Такая запись означает: "записать в указатель pI адрес переменной i ". Запись &i обозначает адрес переменной i .
Получить значение по этому адресу	<pre>float f, *pF; ... f = *pF;</pre>	Такая запись означает: "записать в переменную f то вещественное число, на которое указывает указатель pF ". Запись *pF обозначает содержимое ячейки, на которую указывает pF .
Сдвинуть	<pre>int *pI; ... pI++; pI += 4; -- pI; pI -= 4;</pre>	В результате этих операций значение указателя меняется особым способом: pI++ сдвигает указатель на РАЗМЕР ЦЕЛОГО ЧИСЛА, то есть на 2 байта, а не на 1 как можно подумать. А запись pI+=4 или pI=pI+4 сдвигает указатель на 4 целых числа дальше, то есть на 8 байт.
Обнулить	<pre>char *pC; ... pC = NULL;</pre>	Указатель указывает на фиктивный нулевой адрес, который обычно говорит о том, что указатель недействительный. По нему нельзя ничего записывать (это вызывает зависание компьютера).
Вывести на экран	<pre>int i, *pI; ... pI = &i; printf("Адр.i =%p", pI);</pre>	Для вывода указателей используется формат %p .

Итак, что надо знать про указатели:

- указатель - это переменная, в которой записан адрес другой переменной
- при объявлении указателя надо указать тип переменных, на которых он будет указывать, а перед именем поставить знак *****
- знак **&** перед именем переменной обозначает ее адрес
- знак ***** перед указателем в рабочей части программы (не в объявлении) обозначает значение ячейки, на которую указывает указатель
- нельзя записывать по указателю, который указывает непонятно куда - это вызывает сбой компьютера
- для обозначения недействительного указателя используется константа **NULL**
- при изменении значения указателя на **n** он в самом деле сдвигается к **n**-ому следующему числу данного типа, то есть для указателей на целые числа на **n*sizeof(integer)** байт
- указатель печатаются по формату **%p**

Теперь вам должно быть понятно, что многие непонятные функции ввода типа `scanf` и `fscanf` в самом деле принимают в параметрах адреса переменных, например

```
scanf ( "%d", &i);
```



Динамическое выделение памяти

Динамическими называются массивы, размер которых неизвестен на этапе написания программы. Прием, о котором мы будем говорить, относится уже не к стандартному языку Си, а к его расширению Си++. Существуют и стандартные способы выделения памяти в языке Си (с помощью функций `malloc` и `calloc`), но они не очень удобны.

Следующая простейшая программа, которая использует динамический массив, вводит с клавиатуры размер массива, все его элементы, а затем сортирует их и выводит на экран.

#include <stdio.h> void main() {	объявить указатель для массива
int *pI, N;	
printf ("Размер массива > "); scanf ("%d", &N);	выделить память
pI = new int [N];	
if (pI == NULL) { printf("Не удалось выделить память"); return; }	обработка ошибки
for (i = 0; i < N; i ++) { printf ("\nA[%d] > ", i+1); scanf ("%d", &pI[i]);	а можно и так: scanf ("%d", pI+i);
... // здесь сортировка и вывод на экран	
delete pI;	освободить память
}	

Итак, мы хотим выделить в памяти место под новый массив целых чисел во время работы программы. Мы уже знаем его размер, пусть он хранится в переменной *N* (это число обязательно должно быть больше нуля). Оператор выделения памяти `new` вернет нам адрес нового выделенного блока и для работы с массивом нам надо где-то его запомнить. Вы уже знаете, что есть специальный класс переменных для записи в них адресов памяти – указатели.

Что новенького?

- динамические массивы используются тогда, когда на момент написания программы размер массива неизвестен
- для того, чтобы работать с динамическим массивом, надо объявить указатель соответствующего типа (в нем будет храниться адрес первого элемента массива)
- когда требуемый размер массива стал известен, надо использовать оператор `new` языка Си ++, указав в скобках размер массива (в программе для краткости нет проверки на положительность *N*):

```
pI = new int[N];
```

- нельзя** использовать оператор `new` при отрицательном или нулевом *N*
- после выделения памяти надо проверить, успешно ли оно прошло - если память выделить не удалось, то значение указателя будет равно **NULL**, использование такого массива приведет к зависанию компьютера
- работа с динамическим массивом, на который указывает **pI**, идет также, как и с обычным массивом размера *N* (помните, что язык Си не следит за выходом за границы массива)
- после использования массива надо освободить выделенную память, вызвав оператор

```
delete pI;
```

- после освобождения памяти значение указателя не изменяется, но использовать его уже нельзя, потому что память освобождена
- учитывая, что при добавлении числа к указателю он сдвигается на заданное число ячеек **ЗАДАННОГО ТИПА**, то следующие записи равносильны и вычисляют адрес *i*-ого элемента массива:

<code>&pI[i]</code>	и	<code>pI+i</code>
-------------------------	---	-------------------

Отсюда следует, что **pI** совпадает с **&pI[0]**, и поэтому имя массива можно использовать как адрес его начального элемента

- память выделяется в свободной области, которая называется *куча* (*heap*)
- состояние кучи можно проверить с помощью функции `heapcheck` так:

```
#include <alloc.h>
#include <stdlib.h>
...
if ( heapcheck() == _HEAPCORRUPT ) {
    puts("Куча испорчена");
    exit(1);
}
```

аварийный выход с
кодом ошибки 1

- куча может быть испорчена, если был выход за границы массива или удаление неправильного указателя



Ошибки, связанные с выделением памяти

Самые тяжелые и трудно вылавливаемые ошибки в программах на языке Си связаны именно с неверным использованием динамических массивов. В таблице перечислены наиболее тяжелые случаи и способы борьбы с ними.

Ошибка	Причина и способ лечения
Запись в чужую область памяти	Память была выделена неудачно, а массив используется. Вывод: надо всегда делать проверку указателя на NULL .
Повторное удаление указателя	Массив уже удален и теперь удаляется снова. Вывод: если массив удален из памяти, обнулите указатель - ошибка быстрее выявится.
Выход за границы массива	Запись в массив в элемент с отрицательным индексом или индексом, выходящим за границу массива
Утечка памяти	Неиспользуемая память не освобождается. Если это происходит в функции, которая вызывается много раз, то ресурсы памяти скоро будут израсходованы. Вывод: убирайте за собой мусор.



Выделение памяти для матрицы

Для выделения памяти под одномерный массив целых чисел нам потребовался указатель на целые числа. Для матрицы надо выделить указатель на массив целых чисел, который объявляется как **int **p;**. Но лучше всего сразу объявить новый тип данных - указатель на целое число. Новые типы объявляются директивой **typedef** вне всех процедур и функций (там же, где и глобальные переменные).

```
typedef int *pInt;
```

Этой строкой мы сказали компилятору, что любая переменная нового типа **pInt** представляет собой указатель на целое число или адрес массива целых чисел. К сожалению, место для матрицы не удастся так же просто выделить в памяти, как мы делали это для одномерного массива. Если написать просто

```
int M = 5, N = 7;
pInt *p;
p = new int[M][N];
```

компилятор выдает множество ошибок. Связано это с тем, что ему требуется заранее знать длину одной строки, чтобы правильно расшифровать запись типа **p[i][j]**. Ниже рассмотрены три способа решения этой проблемы.



Известный размер строки

Если размер строки матрицы известен, а неизвестно только количество строк, можно поступить так: ввести новый тип данных – строка матрицы. Когда количество строк станет известно, с помощью оператора **new** выделяем массив таких данных.

<code>typedef int row10[10];</code>	любая переменная типа row10 - это массив из 10 целых чисел
<code>void main()</code> <code>{</code> <code>int N;</code>	
<code>row10 *p;</code>	указатель на массив из элементов типа row10
<code>printf ("Введите число строк ");</code> <code>scanf ("%d", &N);</code>	
<code>p = new row10[N];</code>	выделить память на <i>N</i> строк
<code>p[0][1] = 25;</code> <code>...</code> <code>printf("%d", p[2][3]);</code>	используем матрицу
<code>delete p;</code> <code>}</code>	освобождаем память



Неизвестный размер строки

Пусть размеры матрицы **M** и **N** заранее неизвестны и определяются в ходе работы программы. Тогда можно предложить следующий способ выделения памяти под новую матрицу. Поскольку матрицу можно рассматривать как массив из строк-массивов, объявим **M** указателей и выделим на каждый из них область памяти для одномерного массива размером **N** (то есть, на одну строку). Сами эти указатели тоже надо представить в виде динамического массива. Определив требуемые размеры матрицы, мы выделяем сначала динамический массив указателей, а потом на каждый указатель - место для одной строки.

<code>typedef int *pInt;</code>	
<code>void main()</code> <code>{</code> <code>int M, N, i;</code>	объявили указатель на указатель
<code>pInt *p;</code>	
<code>... // ввод M и N</code>	
<code>p = new pInt[M];</code>	выделяем память под массив указателей
<code>for (i = 0; i < M; i ++)</code> <code>p[i] = new int[N];</code>	выделяем память под строки
<code>... // работаем с матрицей</code>	
<code>for (i = 0; i < M; i ++)</code> <code>delete p[i];</code>	освобождаем память строк
<code>delete p;</code> <code>}</code>	освобождаем массив указателей

Такой способ имеет один недостаток - на каждую строку выделяется свой участок памяти. Учитывая, что память выделяется блоками по 16 байт (параграфами), это может привести к лишнему расходу памяти. Выход из этой ситуации такой – сначала выделим область памяти сразу на весь массив и запишем ее адрес в **p[0]**. Затем расставим указатели так, чтобы **p[1]** указывал на **N+1**-ый элемент с начала блока (начало строки 1), **p[2]** - на **2N+1**-ый (начало строки 2) и т.д. Таким образом, в памяти выделяется всего два блока – на массив указателей и на саму матрицу.

typedef int *pInt;	
void main() { int M, N, i; pInt *p;	объявили указатель на указатель
... // ввод M и N	выделяем память под массив указателей
p = new pInt[M]; p[0] = new int [M*N]; for (i = 1; i < M; i ++) p[i] = p[i-1] + N;	выделяем память на весь массив
... // работаем с матрицей	каждый указатель, начиная с p[1] , сдвигаем на N элементов по сравнению с предыдущим
delete p[0]; delete p;	освобождаем основной массив
}	освобождаем массив указателей

6. Рекурсия



Что такое рекурсия?

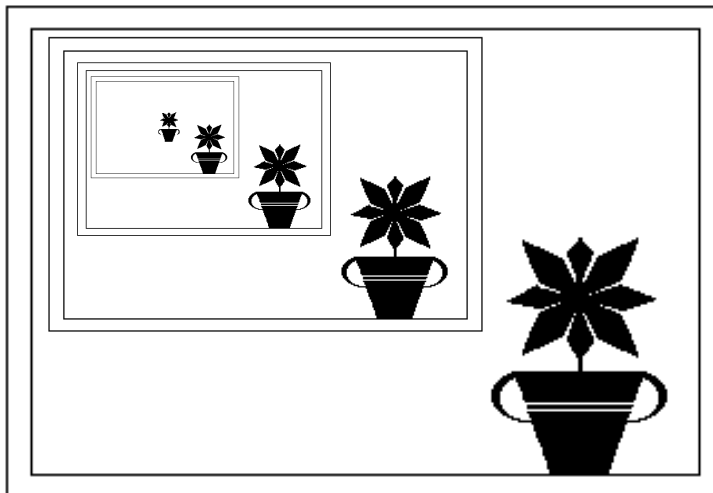


Рекурсивные объекты

Всем известна сказка, которая начинается словами "У попа была собака ...". Это бесконечное предложение будем называть сказкой о попе и собаке. Как же нам определить его строго математически? Оказывается, это можно сделать примерно так:

У попа была собака, он ее любил.
Она съела кусок мяса, он ее убил.
В ямку закопал, надпись написал:
Сказка о попе и собаке

Как видите, частью этой сказки является сама сказка. Так мы пришли к понятию рекурсии. Еще один пример - картинка, содержащая свое изображение.



Рекурсия - это определение объекта через самого себя.

С помощью рекурсии в математике определяются многие бесконечные множества, например множество натуральных чисел. Действительно, можно задать их так:

Натуральное число.

- 1) 1 - натуральное число.
- 2) Число, следующее за натуральным - натуральное.

Понятие факториала также можно задать рекурсивно:

Факториал.

- 1) $0! = 1$ (так условно принято).
- 2) $n! = n * (n-1)!$

Рекурсивные процедуры и функции

Причем же здесь программирование? Оказывается, процедуры и функции в современных языках программирования также могут вызывать сами себя.

Рекурсивными называются процедуры и функции, которые вызывают сами себя.

Например, функцию вычисления факториала можно записать так:

```
int Factorial ( int n )
{
    if ( n <= 0 ) return 1;
    else
        return n*Factorial(n-1);
}
```

рекурсия закончилась

рекурсивный вызов

Обратите внимание, что функция `Factorial` вызывает сама себя, если $n > 0$. Для решения этой задачи можно использовать и рекурсивную процедуру (а не функцию). Вспомним, как рекурсивная процедура может вернуть значение-результат? Через параметр, переданный по ссылке (в объявлении процедуры у его имени стоит знак ссылки `&`). При рекурсивных вызовах процедура меняет это значение.

```
void Factorial ( int n, int &fact )
{
    if ( n == 0 ) fact = 1;
    else {
        Factorial ( n-1, fact );
        fact *= n;
    }
}
```

рекурсия закончилась

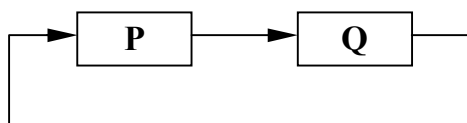
рекурсивный вызов:
вычисляем $(n-1)!$

вычислим $n! = n * (n-1)!$

В отличие от функции, процедура может (если надо) возвращать несколько значений-результатов с помощью параметров, передаваемых по ссылке.

Косвенная рекурсия

Реже используется более сложная конструкция, когда процедура вызывает сама себя не напрямую, а через другую процедуру (или функцию). Это описывается следующей схемой:



Такой прием называется *косвенная рекурсия*.

Не допустим бесконечную рекурсию!

При использовании рекурсивных процедур и функций велика опасность, что вложенные вызовы будут продолжаться бесконечно (это похоже на заикливание цикла `while`). Поэтому основное требование состоит в том, что в таких функциях необходимо предусмотреть условие, которое проверяется на каждом шаге и заканчивает вложенные вызовы, когда перестает выполняться.

Для функции вычисления факториала таким условием является $n \leq 0$. Докажем, что рекурсия в примере, рассмотренном выше, закончится.

1. Рекурсивные вызовы заканчиваются, когда n становится равно нулю.
2. При каждом новом рекурсивном вызове значение n уменьшается на 1 (это следует из того, что вызывается `Factorial(n-1, ...)`).

3. Поэтому, если в начале $n > 0$, то, постепенно уменьшаясь, n достигает значения 0 и рекурсия заканчивается.

Рекурсивная процедура или функция должна содержать условие, при котором рекурсия заканчивается (не происходит нового вложенного вызова)



Когда рекурсия не нужна

При новом рекурсивном вызове компьютер делает так:

1. Запоминается состояние вычислений на данном этапе рекурсивных вызовов.
2. В стеке (особой области памяти) создается **новый** набор локальных переменных (чтобы не испортить переменные текущего вызова).

Поскольку при каждом вызове затрачивается новая память и расходуется время на вызов процедуры и возврат из нее, при использовании рекурсии необходимо помнить, что

глубина рекурсии (количество вложенных вызовов) должна была достаточно мала

Программа, использующая рекурсию с большой глубиной, будет выполняться долго и может вызвать **переполнение стека** (нехватку стековой памяти). Поэтому

если задача может быть легко решена без использования рекурсии, рекурсию использовать нежелательно

Например, задача вычисления факториала очень просто решается с помощью обычного цикла for (такое решение с помощью циклов называют *итеративным*):

```
int Factorial ( int n )
{
    int i, fact = 1;
    for ( i = 2; i <= n; i ++ )
        fact *= i;
    return fact;
}
```

Эта функция работает намного быстрее, чем рекурсивная.

Доказано, что любая рекурсивная программа может быть написана без использования рекурсии, хотя такая реализация может оказаться очень сложной.

Задача. Составить функцию для вычисления чисел Фибоначчи f_i , которые задаются так:

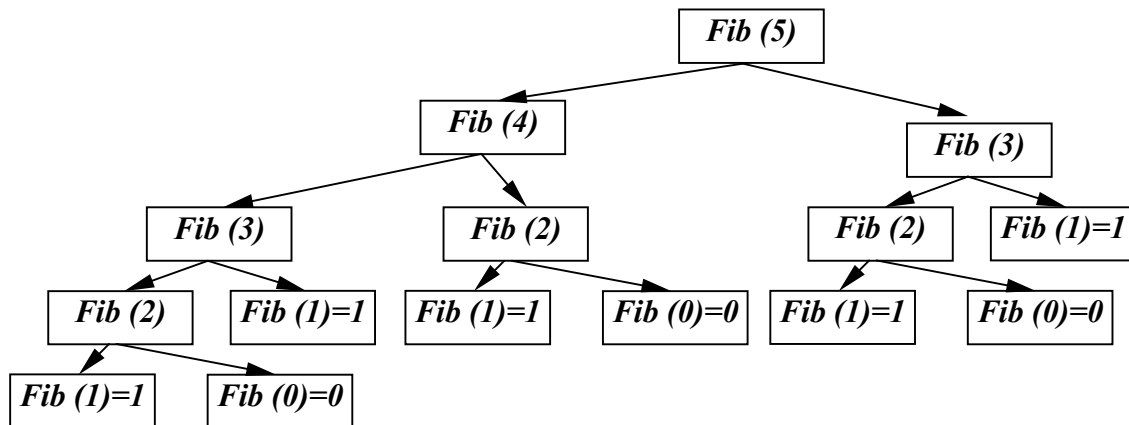
1. $f_0 = 0, f_1 = 1$.
2. $f_i = f_{i-1} + f_{i-2}$ для $i > 1$.

Использование рекурсии "в лоб" дает функцию

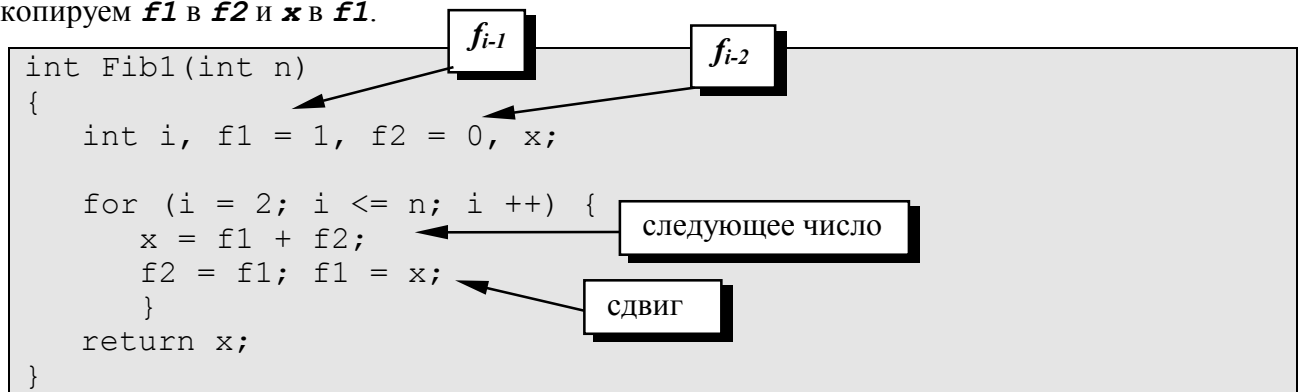
```
int Fib ( int n )
{
    if ( n == 0 ) return 0;
    if ( n == 1 ) return 1;
    return Fib(n-1) + Fib(n-2);
}
```

Заметим, что каждый рекурсивный вызов при $n > 1$ порождает еще 2 вызова функции, многие выражения (числа Фибоначчи для малых n) вычисляются много раз. Поэтому практического

значения этот алгоритм не имеет, особенно для больших n . Схема вычисления **Fib(5)** показана в виде дерева ниже:



Заметим, что очередное число Фибоначчи зависит только от двух предыдущих, которые будем хранить в переменных **f1** и **f2**. Сначала примем **f1=1** и **f2=0**, затем вычисляем следующее число Фибоначчи и записываем его в переменную **x**. Теперь значение **f2** уже не нужно и мы скопируем **f1** в **f2** и **x** в **f1**.

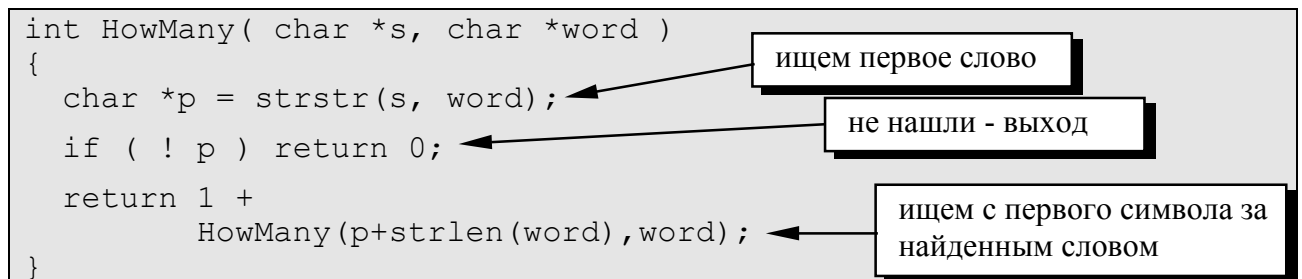


Такая процедура для больших n (>20) работает в сотни тысяч (!!!) раз быстрее, чем рекурсивная. Вывод - там, где можно легко обойтись без рекурсии, надо без нее обходиться.

Рекурсивный поиск

Вспомним задачу, которая рассматривалась при обсуждении работы со строками: определить, сколько раз встречается заданное слово в предложении. Рекурсивную процедуру можно описать так:

1. Ищем первое заданное слово с помощью функции `strstr`. Если не нашли, то стоп.
2. Количество слов = 1 + количество слов в оставшейся части строки.

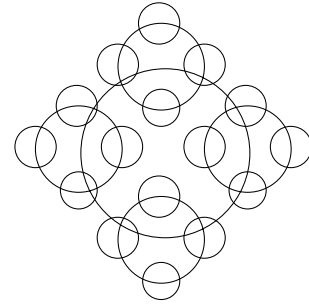


Функция получилась короткая, понятная, но по скорости работы не самая эффективная.



Рекурсивные фигуры

Многие рисунки и кривые задаются рекурсивно. Рассмотрим сначала простейший пример. В центре рисунка - большая окружность. На ней находятся центры четырех окружностей меньшего диаметра, на каждой из которых - центры еще четырех окружностей еще меньшего диаметра и т.д. Всего - n разных диаметров (n уровней рекурсии).



```
void RecCircle ( float x, float y, float R,
int n )
{
    float k = 0.5;
    circle ( x, y, R );
    if ( n == 1 ) return;
    RecCircle( x+R, y, k*R, n-1);
    RecCircle( x-R, y, k*R, n-1);
    RecCircle( x, y+R, k*R, n-1);
    RecCircle( x, y-R, k*R, n-1);
}
```

рисует окружность

больше рисовать не надо

рекурсивные вызовы

Процедура для рисования такой фигуры принимает 4 параметра - координаты центра базовой окружности (x , y), ее радиус R и количество уровней n , которые надо нарисовать. На следующем уровне радиус изменяется в k раз (в примере - $k=0.5$), количество оставшихся уровней уменьшается на 1 и пересчитываются координаты для 4-х окружностей следующего уровня.

Важно доказать, что рекурсия закончится. В самом деле, как видно из процедуры, при $n=1$ рекурсия заканчивается. При каждом новом рекурсивном вызове количество уровней n уменьшается на 1, поэтому если в самом начале $n>0$, то оно достигнет значения 1 и рекурсия завершится. Для большей "защиты от дурака" лучше условие окончания рекурсии записать так:

```
if ( n <= 1 ) return;
```

Если этого не сделать, рекурсия никогда не завершится (теоретически), если в начале задали $n<1$.

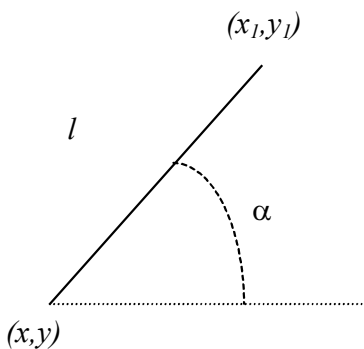
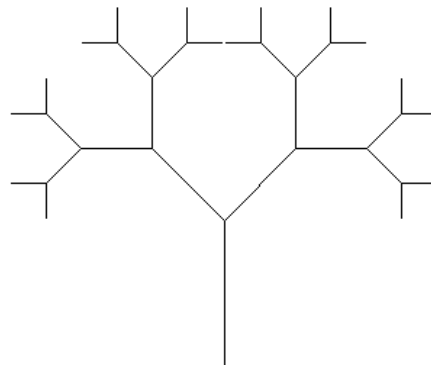
Основная программа получается очень короткой - в ней надо включить монитор в нужный графический режим, один раз вызвать процедуру RecCircle, сделать паузу до нажатия на любую клавишу и выключить графический режим.

```
#include <conio.h>
#include <graphics.h>
... // сюда надо вставить процедуру
void main()
{
    int d = VGA, m = VGAHI;
    initgraph(&gd, &gm, "c:\\borlandc\\bgi");
    RecCircle ( 320, 240, 100, 3 );
    getch();
    closegraph();
}
```

3 уровня

Пифагорово дерево

Пифагорово дерево задается так: "ствол" дерева - вертикальный отрезок длиной 1 . От него вверх симметрично отходят две ветки так, чтобы угол между ними был 90 градусов, их длина равна $k*1$, где $k < 1$. Так повторяется для заданного количества уровней. На рисунке показано дерево Пифагора при $k=0.7$, имеющее 5 уровней.



Чтобы написать процедуру, придется вспомнить геометрию и тригонометрию. Проще всего получается, если написать формулы для произвольного угла наклона ветки α . Если (x, y) - координаты начала ветки и она наклонена по отношению к горизонту на угол α , то формулы для определения координат конца ветки (x_1, y_1) принимают вид:

$$x_1 = x + l * \cos(\alpha)$$

$$y_1 = y - l * \sin(\alpha)$$

Знак минус при вычислении y_1 появился из-за того, что ось y для экрана направлена вниз. Ветки следующего уровня, выходящие из вершины с координатами (x_1, y_1) , имеют углы наклона $\alpha + \pi/4$ (правая ветка) и $\alpha - \pi/4$ (левая ветка). Не забудем также, что для использования функций \sin и \cos надо подключить заголовочный файл `math.h` и передавать им значения углов в радианах. Параметрами процедуры являются координаты начала базовой ветки (x, y) , ее длина l , угол наклона в радианах **angle** и количество уровней **n**, которые надо нарисовать.

```
void Pifagor( float x, float y, float l,
              float angle, int n)
{
    float k = 0.7, x1, y1;
    x1 = x + l*cos(angle);
    y1 = y - l*sin(angle);
    line (x, y, x1, y1);
    if ( n <= 1) return;
    Pifagor(x1, y1, k*l, angle+M_PI_4, n-1);
    Pifagor(x1, y1, k*l, angle-M_PI_4, n-1);
}
```

координаты второго конца

рисуем ветку

больше рисовать не надо

рекурсивные вызовы

Основная программа выглядит примерно так же, как и в предыдущем примере.

Перебор вариантов

Одна из главных практически важных областей, где применение рекурсии жизненно необходимо и значительно упрощает решение – задачи на перебор вариантов.

Сочетания

Необходимо получить все возможные сочетания чисел от 1 до K , которые имеют длину N (в последовательности могут быть одинаковые элементы). Для решения задачи выделим в памяти глобальный массив **A[N]**. Представим себе, что все его первые q элементов (с номерами

от 0 до $q-1$) уже определены и надо найти все сочетания, при которых эти элементы не меняются.

Сначала ставим на место элемент с номером q . По условию на этом месте может стоять любое число от 1 до K , поэтому сначала ставим 1 и находим все варианты, при которых $q+1$ элементов уже поставлены (здесь будет рекурсивный вызов), затем ставим на это место 2 и т.д. Если $q=N$, то все элементы уже поставлены на место и надо печатать массив - это одна из нужных комбинаций. Ниже приведена рекурсивная процедура, которая использует массив $A[N]$ и печатает все комбинации на экране.

```
void Combinations ( int A[], int N, int K, int q )
{
    if ( q == N )           ← комбинация получена
        PrintData ( A, N );
    else
        for (int i = 1; i <= K; i ++ ) {
            A[q] = i;
            Combinations(A, N, K, q+1); ← рекурсивный вызов
        }
}
```

Для вывода массива на экран используется такая процедура:

```
void PrintData ( int Data[], int N )
{
    for (int i = 0; i < N; i++ )
        printf("%2d ", Data[i]);
    printf("\n");
}
```

В основной программе надо вызвать процедуру с параметром $q=0$, поскольку ни один элемент еще не установлен:

```
#include <stdio.h>
... // здесь надо разместить процедуры
void main()
{
    int A[5], N = 5, K = 10;
    Combinations ( A, N, K, 0 );
}
```



Перестановки

Существует еще одна весьма непростая задача, которая хорошо решается с помощью рекурсии. Представьте, что к вам пришли N гостей. Сколько существует различных способов рассадить их за столом?

Сформулируем условие задачи математически. В массиве $A[N]$ записаны целые числа. Надо найти все возможные перестановки этих чисел (в перестановку должны входить **все** элементы массива по 1 разу).

Как и в предыдущем примере, предположим, что q элементов массива (с номерами от 0 до $q-1$) уже стоят на своих местах. Тогда в позиции q может стоять любой из неиспользованных элементов - их надо перебрать в цикле и вызвать рекурсивно эту же процедуру, увеличив на 1 количество установленных элементов. Чтобы не потерять никакой элемент, в цикле для всех i от q до $N-1$ будем переставлять элементы $A[i]$ и $A[q]$, генерировать все возможные комбинации, а затем менять их местами, восстанавливая исходный порядок.

Для обмена местами двух элементов массива будем использовать вспомогательную переменную *temp*.

<pre>void Permutations (int A[], int N, int q) { int temp; if (q == N)</pre>		вывод перестановки на экран
<pre> PrintData (A, N);</pre>		
<pre> else for (int i = q; i <= N; i ++) {</pre>		меняем местами A[q] и A[i]
<pre> temp = A[q]; A[q] = A[i]; A[i] = temp;</pre>		
<pre> Permutations(A, N, q+1);</pre>		рекурсивный вызов
<pre> temp = A[q]; A[q] = A[i]; A[i] = temp; }</pre>		ставим A[i] обратно
<pre> }</pre>		



Разложение числа на сумму слагаемых

Задача. Дано целое число *S*. Требуется получить и вывести на экран все возможные **различные** способы представления этого числа в виде суммы натуральных чисел (то есть, 1 + 2 и 2 + 1 - это одинаковый способ разложения числа 3).

Самое сложное в этой задаче – сразу исключить из рассмотрения повторяющиеся последовательности. Для этого будем рассматривать только *невозрастающие* последовательности слагаемых (каждое следующее слагаемое *не больше* предыдущего).

Все слагаемые будем записывать в массив. Для этого нам понадобится массив (назовем его **A**) из *S* элементов, поскольку самое длинное представление - это сумма *S* единиц. Чтобы не расходовать память зря, лучше выделять его динамически (см. раздел про массивы).

Пусть *q* элементов массива (с номерами от 0 до *q-1*) уже стоят на своих местах, причем **A[q-1]=m**. Каким может быть следующее слагаемое - элемент **A[q]**? Поскольку последовательность невозрастающая, он не может быть больше *q*. С другой стороны, он должен быть не больше, чем *S-S₀*, где *S₀* - сумма предыдущих элементов. Очевидно, что минимальное значение этого элемента - единица.

Эти размышления приводят к следующей процедуре, которая принимает в параметрах сумму оставшейся части, массив для записи результата и количество уже определенных элементов последовательности.

<pre>void Split(int R, int A[], int q) { int i, last = R; if (R == 0) PrintData (A, q); else {</pre>		
<pre> if (q > 0 && last > A[q-1])</pre>		вычислить наибольшее возможное A[q]
<pre> last = A[q-1];</pre>		
<pre> for (i = 1; i <= last; i ++) {</pre>		перебрать все A[q] от 1 до <i>last</i>
<pre> A[q] = i;</pre>		
<pre> Split (R-i, A, q+1);</pre>		новый остаток
<pre> }</pre>		
<pre> }</pre>		

Основная программа вызывает эту процедуру так (используется динамический массив):

```
#include <stdio.h>
... // здесь надо поместить процедуры Split и PrintData
void main()
{
    int *A, S;
    printf ( "Введите натуральное число " );
    scanf ( "%d", &S );
    A = new int[S];
    Split (S, A, 0);
    delete A;
}
```



Быстрая сортировка

Один из лучших известных методов сортировки массивов - **быстрая сортировка** Ч. Хоара (*Quicksort*) основана как раз на применении рекурсии.

Будем исходить из того, что сначала лучше делать перестановки элементов массива на большом расстоянии. Предположим, что у нас есть n элементов и известно, что они уже отсортированы в обратном порядке - тогда за $n/2$ обменов можно отсортировать их как надо - сначала поменять местами первый и последний, а затем последовательно двигаться с двух сторон. Хотя это справедливо только тогда, когда порядок элементов в самом деле обратный, подобная идея заложена в основу алгоритма *Quicksort*.

Пусть дан массив **A** из n элементов. Выберем сначала наугад любой элемент массива (назовем его **x**). Обычно выбирают средний элемент массива, хотя это не обязательно. На первом этапе мы расставим элементы так, что слева от некоторой границы находятся все числа, меньшие или равные **x**, а справа – большие или равные **x**. Заметим, что элементы, равные **x**, могут находиться в обеих частях.

$A[i] \leq x$	$A[i] \geq x$
---------------	---------------

Теперь элементы расположены так, что ни один элемент из первой группы при сортировке не окажется во второй и наоборот. Поэтому далее достаточно отсортировать отдельно каждую часть массива.

Размер обеих частей чаще всего не совпадает. Лучше всего выбирать **x** так, чтобы в обеих частях было равное количество элементов. Такое значение **x** называется *медианой* массива. Однако для того, чтобы найти такое **x**, надо сначала отсортировать массив, то есть заранее решить ту самую задачу, которую мы собираемся решить этим способом. Поэтому обычно в качестве **x** выбирают средний элемент массива.

Будем просматривать массив слева до тех пор, пока не обнаружим элемент, который больше **x** (и, следовательно, должен стоять справа от **x**), потом справа пока не обнаружим элемент меньше **x** (он должен стоять слева от **x**). Теперь поменяем местами эти два элемента и продолжим просмотр до тех пор, пока два просмотра не встретятся где-то в середине массива. В результате массив окажется разбитым на 2 части: левую со значениями меньшими или равными **x**, и правую со значениями большими или равными **x**. Затем такая же процедура применяется к обеим частям массива до тех пор, пока в каждой части не останется один элемент (и таким образом, массив будет отсортирован).

Чтобы понять сущность метода, рассмотрим пример. Пусть задан массив

78	6	82	67	55	44	34
----	---	----	----	----	----	----

x

Выберем в качестве **x** средний элемент массива, то есть **67**. Найдем первый слева элемент массива, который больше или равен **x** и должен стоять во второй части. Это число **78**. Обозначим номер этого элемента через **i**. Теперь находим первый справа элемент, который меньше **x** и должен стоять в первой части. Это число **34**. Обозначим его номер через **j**.

78	6	82	67	55	44	34
----	---	----	----	----	----	----

i**j**

Теперь поменяем местами два этих элемента. Сдвигая переменную **i** вправо, а **j** – влево, находим следующую пару, которую надо переставить. Это числа **82** и **44**.

34	6	82	67	55	44	78
----	---	----	----	----	----	----

i**j**

Следующая пара элементов – числа 67 и 55.

34	6	44	67	55	82	78
----	---	----	----	----	----	----

После этой перестановки дальнейший поиск приводит к тому, что переменная **i** становится больше **j**, то есть массив разбит на две части. После такой операции все элементы массива, расположенные левее **A[i]**, меньше **x**, а все правее **A[j]** – больше **x**. В данном примере средний элемент остается на своем месте, но в других случаях он может быть переставлен.

34	6	44	55	67	82	78
----	---	----	----	----	----	----

Теперь остается применить тот же способ рекурсивно к первой и второй частям массива. Рекурсия заканчивается, когда в какой-то части остается один элемент. Этот алгоритм реализован в процедуре, приведенной ниже.

```

void QuickSort ( int A[], int from, int to )
{
    int x, i, j, temp;
    if ( from >= to ) return; ← условие окончания рекурсии
    i = from;
    j = to; ← рассматриваем только элементы с A[from] по A[to]
    x = A[(from+to)/2]; ← выбрали средний элемент в этом интервале

    while ( i <= j ) {
        while ( A[i] < x ) i ++;
        while ( A[j] > x ) j --; ← ищем пару для перестановки

        if ( i <= j ) {
            temp = A[i]; A[i] = A[j]; A[j] = temp;
            i ++;
            j --; ← переставляем элементы
        } ← двигаемся дальше
    }

    QuickSort ( A, from, j );
    QuickSort ( A, i, to ); ← сортируем две оставшиеся части
}

```

Насколько эффективна такая сортировка? Все описанные ранее способы сортировки массивов имели порядок $O(n^2)$, то есть при увеличении вдвое длины массива время сортировки увеличивается вдвое. Можно показать, что **Quicksort** имеет в среднем порядок $O(n \log n)$, что значительно лучше.

Чтобы почувствовать, что такое $O(n \log n)$ и $O(n^2)$, посмотрите на таблицу, где приведены числа, иллюстрирующие скорость роста для нескольких разных функций.

n	$\log n$	$n \log n$	n^2
1	0	0	1
16	4	64	256
256	8	2 048	65 536
4 096	12	49 152	16 777 216
65 536	16	1 048 565	4 294 967 296
1 048 476	20	20 969 520	1 099 301 922 576
16 775 616	24	402 614 784	281 421 292 179 456

Если считать, что числа в таблице соответствуют микросекундам, то для задачи с 1048476 элементами алгоритму со временем работы $O(\log n)$ потребуется 20 микросекунд, а алгоритму с временем работы $O(n^2)$ – более 12 дней.

Однако, этому алгоритму присущи и недостатки. Все зависит от того, насколько удачным будет на каждом шаге выбор x . В идеальном случае мы должны всегда выбирать в качестве x медиану – такой элемент, что в данном диапазоне есть равное количество меньших и больших элементов. При этом каждый раз зона сортировки сокращается в два раза и требуется всего $\log n$ проходов, общее число сравнений равно $n \log n$. В самом худшем случае (если мы каждый раз наибольшее значение из данного диапазона) он имеет скорость $O(n^2)$.

Кроме того, этот алгоритм, как и все усовершенствованные методы сортировки, неэффективен при малых n . Иногда для сортировки коротких отрезков в него включают прямые методы сортировки.

Ниже приведены результаты теста – время сортировки в секундах для разных наборов данных тремя способами (для процессора **Pentium 120**).

n	тип данных	метод пузырька	метод выбора мин. элемента	быстрая сортировка
1000	возрастающие	0.199	0.104	0.003
	убывающие	0.226	0.123	0.003
	случайные	0.235	0.120	0.004
5000	возрастающие	4.698	2.898	0.023
	убывающие	5.027	2.788	0.024
	случайные	5.275	2.953	0.024
15000	возрастающие	41.319	27.143	0.080
	убывающие	44.945	34.176	0.084
	случайные	44.780	33.571	0.068

Таким образом, для массива размером 15000 элементов метод **Quicksort** более чем в 500 раз эффективнее, чем прямые методы.

7. Структуры

Что такое структуры?

Очень часто при обработке информации приходится работать с блоками данных, в которых присутствуют разные типы данных. Например, информация о книге в каталоге библиотеки включает в себя автора, название книги, год издания, количество страниц и т.п.

Для хранения этой информации в памяти не подходит обычный массив, так как в массиве все элементы должны быть одного типа. Конечно, можно использовать несколько массивов разных типов, но это не совсем удобно (желательно, чтобы все данные о книге находились в памяти в одном месте).

В современных языках программирования существует особый тип данных, который может включать в себя несколько элементов более простых (причем разных!) типов.

Структура - это тип данных, который может включать в себя несколько **полей** - элементов разных типов (в том числе и другие структуры).

Одно из применений структур –организация различных баз данных, списков и т.п.

Объявление и инициализация

Поскольку структура - это новый тип данных, его надо предварительно объявить в начале программы, например так

```
struct Book {  
    char author[40];    // автор, символьная строка  
    char title[80];     // название, символьная строка  
    int year;           // год издания, целое число  
    int pages;          // количество страниц, целое число  
};
```

Не забудем, что при таком объявлении типа никакая память не выделяется, реальных структур в памяти еще нет. Этот фрагмент просто скажет компилятору, что в программе МОГУТ БЫТЬ элементы такого типа. Теперь, когда надо выделить память, мы напишем

```
Book b;
```

Это оператор выделяет место в памяти под одну структуру типа **Book** и дает ей имя **b** (таким образом, **b** – это имя конкретного экземпляра). При этом можно сразу записать в память нужное начальное значение всех или нескольких первых элементов структуры в фигурных скобках через запятую. Поля заполняются последовательно в порядке их объявления. Память, отведенная на поля, оставшиеся незаполненными, обнуляется.

```
Book b = {  
    "А. С. Пушкин",  
    "Полтава",  
    1998 };
```



Работа с полями структуры



Обращение по имени

Для обращения ко всей структуре используется ее имя, а для обращения к отдельному полю имя этого поля ставится через **точку**. Элементы структуры вводятся последовательно по одному. Заполнять их можно в любом порядке. С полем структуры можно работать так же, как и с переменной соответствующего типа: числовые переменные могут участвовать в арифметических выражениях, со строками можно выполнять все стандартные операции.

```
Book b;

strcpy ( b.author, " А.С. Пушкин " );
b.year = 1998;
```



Обращение по адресу

Пусть известен адрес структуры в памяти. Как известно, адрес может быть записан в указатель – специальную переменную для хранения адреса. Для обращения к полю структуры по ее адресу используют специальный оператор **->**.

```
Book b;
Book *p;
p = &b;
strcpy ( p->author, " А.С. Пушкин " );
p->year = 1998;
```

указатель на структуру

записать адрес структуры **b**



Ввод и вывод



Поэлементный ввод и вывод

При вводе с клавиатуры и выводе на экран или в текстовый файл с каждым полем структуры надо работать отдельно, как с обычной переменной. В приведенном примере данные вводятся в структуру типа `Book` с клавиатуры и записываются в файл.

```
Book b;
FILE *fp;

printf("Автор ");
gets(b.author);
printf("Название книги ");
gets(b.title);
printf("Год издания, кол-во страниц ");
scanf("%d%d", &b.years, &b.pages );

fp = fopen("books", "a");
fprintf("%s\n%s\n%d %d\n",
        b.author, b.title, b.years, b.pages );
fclose ( fp );
```

добавить в конец файла

Работа с двоичным файлом

Структуры очень удобно записывать в двоичные файлы, поскольку можно за 1 раз прочитать или записать сразу одну или даже несколько структур. Вспомним, что при чтении из двоичного файла функции `fread` надо передать адрес нужной области памяти (куда записать прочитанные данные), размер одного блока и количество таких блоков. Для того, чтобы не вычислять размер структуры вручную (можно легко ошибиться), применяют оператор `sizeof`.

```
Book b;
int n;
FILE *fp;

fp = fopen("books", "rb");
n = fread (&b, sizeof(Book), 1, fp);
if ( n == 0 ) {
    printf ( "Ошибка при чтении из файла" );
}

fclose ( fp );
```

Можно было также вместо `sizeof(Book)` написать `sizeof(b)`, поскольку запись **b** - это как раз один экземпляр структуры. Функция `fread` возвращает число удачно прочитанных элементов (в нашем случае – структур). Поэтому если в примере переменная **n** равно нулю, чтение закончилось неудачно и надо вывести сообщение об ошибке.

Для записи структуры в двоичный файл используют функцию `fwrite`. Ее параметры – те же, что и у `fread`. Пример ниже показывает добавление структуры в конец двоичного файла.

```
Book b;
FILE *fp;

... // здесь надо заполнить структуру

fp = fopen("books", "ab");
fwrite(&b, sizeof(Book), 1, fp);

fclose ( fp );
```

Копирование

Задача. Пусть в памяти выделено две структуры одного типа и в одну из них записаны какие-то данные. Требуется скопировать все данные из первой структуры во вторую.

Пусть структуры имеют тип **Book** и называются **b1** и **b2**. Существуют три способа решения этой задачи. Самый сложный – копирование каждого поля отдельно:

```
Book b1, b2;

... // здесь заполняем структуру b1

strcpy ( b2.author, b1.author );
strcpy ( b2.title, b1.title );
b2.year = b1.year;
b2.pages = b1.pages;
```

Можно использовать специальную функцию `memcpy`, которая умеет копировать блоки памяти. Для ее использования надо подключить к программе заголовочный файл **mem.h**.

```
#include <mem.h>
```

```
...
```

```
Book b1, b2;
```

```
... // ввод b1
```

```
memcpy ( &b2, &b1, sizeof(Book) );
```

куда (адрес)

откуда (адрес)

сколько байт

Самый простой способ – третий. Достаточно просто написать

```
b2 = b1;
```

При этом будет вызвана функция `memcpy`, которая скопирует одну структуру в другую «бит в бит». Зачем же рассказывать про остальные два способа? Только для того, чтобы понять, как это все работает, поскольку непонимание ведет к трудноуловимым ошибкам.



Массивы структур

Структуры служат для обработки большого объема информации, поэтому чаще всего в программе используются массивы структур. Они объявляются так же, как обычно, но предварительно (выше) надо объявить саму структуру как новый тип данных.

Для обращения к полю структуры также используют точку, но теперь надо указать в квадратных скобках еще номер нужной структуры, например

```
Book A[20];
```

```
...
```

```
A[12].pages = 50;
```

```
for ( i = 0; i < 20; i ++ )
```

```
    puts(A[i].title);
```

вывести названия всех книг

Если вы работаете с двоичными файлами, то чтение и запись всего массива структур выполняется в одну строчку. Покажем приемы работы с двоичным файлом на примере задачи.

Задача. В файле "input.dat" записаны структуры типа **Book**. Известно, что их меньше ста. Требуется прочитать их в память, у всех книг установить 2002 год издания и записать обратно в тот же файл.

Поскольку известно, что структур меньше ста, заранее выделяем в памяти массив на 100 структур.

```
Book b[100];
```

При чтении из файла вызываем пытаемся читать все 100 структур:

```
n = fread ( &b[0], sizeof(Book), 100, fp );
```

Чтобы определить, сколько структур было в действительности прочитано, используем значение **n**, которое возвращает функция `fread` возвращает в качестве результата. Ниже приводится полная программа.

#include <stdio.h>	
struct Book { char author[40]; char title[80]; int year; int pages; };	
void main() { Book b[100]; int i, n; FILE *fp;	
fp = fopen("books", "rb"); n = fread(&b[0], sizeof(Book), 100, fp); fclose (fp);	пытаемся читать 100 структур
for (i = 0; i < n; i ++) b[i].year = 2002;	
fp = fopen("books", "wb"); fwrite (b, sizeof(Book), n, fp); fclose (fp);	записываем столько структур, сколько прочитали
}	записи &b[0] и b равносильны



Динамическое выделение памяти

Предположим, что надо создать массив структур, размер которого выясняется только во время работы программы. Для этого надо

1. Объявить переменную типа указатель на нужный тип данных.
2. Вызвать функцию выделения памяти или оператор `new`.
3. Запомнить в переменной адрес выделенного блока памяти.
4. Использовать новую область как обычный массив.

Book *B; int n;	
printf("Сколько у вас книг? "); scanf ("%d", &n);	сколько структур надо
B = new Book[n]; ... // здесь работаем с массивом B	выделить память
delete A;	освободить память

Иногда требуется выделить в памяти одну структуру. При этом мы получаем ее адрес, который записываем в переменную-указатель. Как при этом получить доступ к полю структуры?

Один из вариантов – "разыменовывать" указатель, то есть обратиться к той структуре, на которую он указывает. Если **p** - указатель на структуру типа **Book**, то обратиться к ее полю **author** можно как **(*p).author**. Эта запись означает "мне нужно поле **author** той структуры, на которую указывает указатель **p**".

В языке Си существует и другой способ обратиться к полю структуры - можно также написать и **p->author**, что значит то же самое, но считается лучшим стилем

программирования. В следующем примере динамически выделяется память на 1 структуру, ее поля считываются с клавиатуры, и затем структура записывается в конец текстового файла.

Book *p; FILE *fp;	выделить память на 1 структуру
p = new Book;	
printf("Автор "); printf("Название книги "); printf("Год издания, кол-во страниц "); scanf("%d%d", &p->years, &p->pages);	ввести данные, обращаясь через указатель
fp = fopen("books", "a");	дописать в конец файла
fprintf("%s\n%s\n%d %d\n", p->author, p->title, p->years, p->pages);	
fclose (fp); delete p;	освободить память



Структуры как параметры процедур

Структуры, так же, как и любые другие типы, могут быть параметрами функций и процедур. В этом разделе будут показаны три способа передачи структур в процедуры и функции и описаны их отличия. Рассмотрим процедуру, которая записывает в поле `year` (год издания книги) значение 2002.



Передача по значению

Если параметр процедуры объявлен как:

```
void Year2002( Book b )
{
    b.year = 2002;
}

void main()
{
    Book b;
    Year2002 ( b );
}
```

то при работе процедуры создается КОПИЯ этой структуры в стеке (специальной области памяти, где хранятся параметры и локальные переменные процедур и функций) и процедура работает с этой копией. Такой подход имеет два недостатка

- Во-первых, процедура не изменяет поля структуры в вызывающей программе. То есть, в нашем случае задача не решается.
- Во-вторых, самое главное – структуры могут быть достаточно больших размеров и создание новой копии может исчерпать стек (по умолчанию он всего 4 Кб).



Передача по ссылке

Поэтому чаще всего параметры-структуры передаются в процедуры и функции *по ссылке* (при объявлении за именем типа структуры ставят знак **&**).

```
void Year2002( Book &b )
{
    b.year = 2002;
}
```

В этом случае фактически в процедуру передается *адрес* структуры и процедура работает с тем же экземпляром, что и вызывающая программа. При этом все изменения, сделанные в процедуре, остаются в силе. Работа со структурой-параметром внутри процедуры и вызов этой процедуры из основной программы никак не отличаются от предыдущего варианта.

Передача по адресу

Передача по ссылке возможна только при использовании языка Си++ (она не входит в стандарт языка Си). Тем не менее, в классическом Си можно передать в качестве параметра *адрес* структуры. При этом обращаться к полям структуры внутри процедуры надо через оператор `->`.

void Year2002(Book *b)	
{	
b->year = 2002;	параметр – адрес структуры
}	обращение к полю структуры
void main()	
{	
Book b;	передается адрес структуры
Year2002 (&b);	
}	

Сортировка по ключу

Поскольку в структуры входит много полей, возникает вопрос: а как их сортировать? Это зависит от конкретной задачи, но существуют общие принципы, о которых мы и будем говорить.

Для сортировки выбирается одно из полей структуры, оно называется *ключевым полем* или просто *ключом*. Структуры расставляются в определенном порядке *по значению ключевого поля*, содержимое всех остальных полей игнорируется.

И еще одна проблема – при сортировке элементы массива меняются местами. Структуры могут быть достаточно больших размеров, и копирование этих структур будет занимать очень много времени. Поэтому

Сортировку массива структур выполняют по указателям.

Мы уже использовали этот прием для сортировки массива символьных строк. В памяти формируется массив указателей *p*, и сначала они указывают на структуры в порядке расположения их в массиве, то есть указатель *p[i]* указывает на структуру с номером *i*. Затем остается только расставить указатели так, чтобы ключевые поля соответствующих структур были отсортированы в заданном порядке. Для решения задачи удобно объявить новый тип данных *PBook* – указатель на структуру *Book*.

```
typedef Book *PBook;
```

Пример ниже показывает сортировку массива структур по году выпуска книг. Основной алгоритм сортировки *массива указателей* заложен в процедуру *SortYear*, которая имеет два параметра – массив указателей и число структур.

```

void SortYear ( PBook p[], int n )
{
    int i, j;
    PBook temp;

    for ( i = 0; i < n-1; i ++ )
        for ( j = n-2; j >= i; j -- )
            if ( p[j+1]->year < p[j]->year )
            {
                temp = p[j];
                p[j] = p[j+1];
                p[j+1] = temp;
            }
}

```

временный указатель

метод пузырька

сравнение ключевых полей

если указатели стоят неправильно, меняем их местами

Основная программа, которая выводит на экран информацию о книгах в порядке их выхода, выглядит примерно так:

```

#include <stdio.h>
const N = 20;

```

```

struct Book {
    char author[40];
    char title[80];
    int year;
    int pages;
};
typedef Book *PBook;

```

новые типы данных

```

... // здесь надо расположить процедуру SortYear
void main()
{

```

```

    Book B[N];
    PBook p[N];

```

массив указателей

```

    ... // здесь ввод данных в структуры

```

```

    for ( i = 0; i < N; i ++ )
        p[i] = &B[i];

```

расставить указатели

```

    SortYear ( p, N );

```

сортировка по указателям

```

    for ( i = 0; i < N; i ++ )
        printf("%d: %s\n", p[i]->year, p[i]->title);

```

вывод через указатели

```

}

```

Глава III.

Разработка программ

Глава III. Разработка программ	1
1. Проектирование программ	2
Этапы разработки программ	2
Программирование «снизу вверх»	4
Структурное программирование	4
2. Построение графиков и работа с ними	7
Структура программы	7
Способы задания функций	8
Системы координат	11
Построение графиков	14
Как найти точки пересечения?	16
Штриховка замкнутой области	21
Площадь замкнутой области	22
3. Вычислительные методы	27
Целочисленные алгоритмы	27
Многоразрядные целые числа	29
Многочлены	31
Последовательности и ряды	32
Численное решение уравнений	36
Вычисление определенных интегралов	40
Вычисление длины кривой	40
Оптимизация	41
4. Моделирование	45
Что такое модель?	45
Виды моделей	45
Вращение	46
Использование массивов	48
Математическое моделирование физических процессов	50
5. Сложные проекты	53
Зачем нужны проекты?	53
Как создать проект?	53
Как загрузить проект?	54
Общие глобальные переменные	54
Общие заголовочные файлы	55
Оверлейные программы	56

1. Проектирование программ



Этапы разработки программ



Постановка задачи

Самое главное – определить цель программы и ее категорию (системная, прикладная, вычислительная и т.д.). Задача может быть сформулирована на естественном языке в свободной форме.

При выполнении коммерческого проекта необходимо в письменной форме оговорить все требования к программе.

Различают **хорошо и плохо поставленные задачи**. В хорошо поставленных задачах четко выделены исходные данные и определены взаимосвязи между ними, позволяющие получить результат.

Задача называется **плохо поставленной**, если для решения не хватает исходных данных, то есть, неясны связи между исходными данными и результатом.

Предположим, Вася купил компьютер. Постепенно его части выходят из строя и ремонт обходится дороже с каждым годом. В то же время, новые компьютеры дешевеют и наступает момент, когда выгоднее не ремонтировать старый компьютер, а продать его и купить новый. Когда Васе наиболее выгодно продать старый компьютер и купить новый?

Эта задача плохо (некорректно) поставлена, поскольку не заданы исходные данные (стоимость нового компьютера, стоимость ремонта каждый год, и т.д.). Вот еще примеры плохо поставленных задач¹:

- Винни Пух и Пятачок построили ловушку для слонopotама. Удастся ли его поймать?
- Винни Пух и Пятачок пошли в гости к Кролику. Сколько бутербродов с медом может съесть Винни, чтобы не застрять в двери?
- Малыш и Карлсон решили по-братски разделить два орешка – большой и маленький. Как это сделать?
- Аббат Фариа решил бежать из замка Иф. Сколько времени ему понадобится для этого?

Однако, если четко определить все существенные свойства, их численные значения и взаимосвязь между этими свойствами и результатом, эти задачи становятся *хорошо поставленными* и могут быть решены.



Разработка модели данных

Одна из основных проблем в практическом программировании – правильно построить формальную модель задачи, определив нужную структуру данных (переменные, массивы, структуры) и взаимосвязи между ними. Большинство наиболее принципиальных и тяжелых ошибок в программах вызваны именно ошибками в выборе модели данных.



Разработка алгоритма

На этом этапе необходимо выбрать или разработать заново **алгоритм**². Алгоритм должен учитывать выбранную ранее модель данных. Иногда приходится возвращаться к предыдущему

¹ А.Г. Гейн и др., Информатика, учебник для 8-9 кл. общеобразоват. учреждений, М.: Просвещение, 1999.

шагу, поскольку для того, чтобы использовать нужный алгоритм, удобнее организовать данные по-другому, например, использовать структуры вместо нескольких отдельных массивов.

Разработка программы

Для того, чтобы исполнитель (например, компьютер) понял и выполнил задание, надо составить его на понятном языке, то есть написать программу³. Важным также является выбор языка программирования. Большинство профессиональных программ сейчас составляется на языке *Си* (и *Си++*), реже – на *Visual Basic* или *Delphi*.

Отладка программы

Отладка – это поиск и исправление ошибок в программе.

На английском языке отладка называется *debugging*, что дословно означает «извлечение жучков». Легенда утверждает, что на заре развития компьютеров (в 1945 году) в контакты электронных реле компьютера MARK-I залетела моль (*bug*), из-за чего он перестал работать.

Для отладки программ используются специальные программы, которые называются **отладчики**. Они позволяют

- выполнять программу в пошаговом режиме (останавливаясь после каждой выполненной команды)
- устанавливать **точки прерывания**, так что программа останавливается каждый раз, когда переходит на заданную строчку
- просматривать и изменять значения переменных в памяти, а также регистры процессора

При отладке можно также использовать **профайлер** – программу, которая позволяет определить, сколько времени выполняется каждая часть программы. При этом надо в первую очередь оптимизировать те процедуры, которые выполняются дольше всех.

Разработка документации

Очень важно правильно разработать документацию, связанную с программой. Чаще всего требуется руководство пользователя (*User manual*), иногда (когда поставляются исходные тексты программы) – руководство программиста, подробное описание алгоритма и особенностей программы. Эти документы надо составлять на простом и понятном языке, так чтобы их мог читать не только автор программы.

Тестирование программы

Тестирование – это проверка программы специальными людьми, которых называют тестеры.

Цель тестирования – найти как можно больше ошибок, которые не смог найти разработчик. Различают две ступени тестирования профессиональных программ:

- **альфа-тестирование** – это тестирование сотрудниками той же фирмы, в которой работает команда разработчиков

² **Алгоритм** – это четко определенная конечная последовательность действия, которую надо выполнить для решения задачи.

³ **Программа** – это алгоритм, записанный на языке программирования. Часто программой называют также любой набор инструкций для компьютера.

- **бета-тестирование** – тестирование предварительной версии в других фирмах и организациях; часто бета-версии программ для тестирования свободно распространяются через Интернет



Сопровождение

Сопровождение – это исправление ошибок в программе после ее сдачи заказчику и консультации для пользователей.

Современный подход к продаже программ сводится к тому, что продается не программа, а техническая поддержка, то есть автор обязуется (за те деньги, которые ему были выплачены) незамедлительно исправлять ошибки, найденные пользователями, и отвечать на их вопросы.



Программирование «снизу вверх»

Раньше этот подход был основным при составлении программ. Мы начинаем составлять программу с самых простых процедур и функций. Таким образом, мы учим компьютер решать все более и более сложные задачи, которые станут частями нашей главной задачи. Построив процедуры нижнего уровня, мы собираем из них, как из кубиков, более крупные процедуры и так далее. Конечная цель – собрать из кубиков всю программу.

Достоинства:

- легко начать составлять программу «с нуля»
- используя метод «от простого – к сложному» можно написать наиболее эффективные процедуры

Недостатки:

- при составлении простейших процедур необходимо как-то связывать их с общей задачей
- может так случиться, что при окончательной сборке не хватит каких-то кубиков, о которых мы не подумали раньше
- программа часто получается запутанной



Структурное программирование



Программирование «сверху вниз»

Более современным считается подход «сверху вниз», когда программа проектируется по принципу «от общего к частному». Этот метод часто называется **методом последовательной детализации**.

Сначала задача разбивается на более простые подзадачи и составляется основная программа. Вместо тех процедур, которые еще не реализованы, пишутся «**заглушки**» - процедуры и функции, которые ничего не делают, но имеют заданный список параметров. Затем каждая подзадача разбивается на еще более мелкие подзадачи и так далее, пока все процедуры не будут реализованы.

Достоинства:

- поскольку мы начинаем с главного – общей задачи, меньше вероятность принципиальной ошибки
- структура программы получается простой и понятной

Недостатки:

- может получиться так, что в разных блоках потребуется реализовать несколько похожих процедур, вместо которых можно было бы обойтись одной

Цели структурного программирования

Структурное появилось потому, что программы становились все сложнее и сложнее, и надо было применять новые методы, чтобы создавать и отлаживать их в короткие сроки. Надо было решить следующие задачи:

- повысить **надежность программ**; для этого нужно, чтобы программа легко поддавалась тестированию и не создавала проблем при отладке.
- повысить **эффективность программ**; сделать так, чтобы текст любого модуля можно было переделать независимо от других для повышения эффективности его работы;
- уменьшить **время и стоимость разработки** сложных программ;
- улучшить **читабельность программ**; это значит, что необходимо избегать использования запутанных команд и конструкций языка; надо разрабатывать программу так, чтобы ее можно было бы читать от начала до конца без управляющих переходов на другую страницу.

Принципы структурного программирования

Принцип абстракции. Программа должна быть спроектирована так, чтобы ее можно было рассматривать с различной степенью детализации, без лишних подробностей.

Принцип модульности. В соответствии с этим принципом программа разделяется на отдельные законченные простые фрагменты (модули), которые можно отлаживать и тестировать независимо друг от друга. В результате отдельные части программы могут создаваться разными группами программистов.

Принцип подчиненности. Взаимосвязь между частями программы должна носить подчиненный характер. К этому приводит разработка программы “сверху вниз”.

Принцип локальности. Надо стремиться к тому, чтобы каждый модуль (процедура или функция) использовал свои (локальные) переменные и команды. Желательно не применять глобальные переменные.

Структурные программы

Программа, разработанная в соответствии с принципами структурного программирования, должна удовлетворять следующим требованиям:

- программа должна разделяться на независимые части, называемые модулями;

Модуль – это независимый блок, код (текст) которого физически и логически отделен от кода других модулей; модуль выполняет только одну логическую функцию, иначе говоря, должен решать самостоятельную задачу своего уровня по принципу: один программный модуль - одна функция;

- работа программного модуля не должна зависеть:
 - ⇒ от того, какому модулю предназначены его выходные данные;
 - ⇒ от того, сколько раз он вызывался до этого;
- размер программного модуля желательно ограничивать одной-двумя страницами исходного листинга (30-60 строк исходного кода);

- модуль должен иметь только одну входную и одну выходную точку;
- взаимосвязи между модулями устанавливаются по принципу подчиненности;
- каждый модуль должен начинаться с комментария, объясняющего его назначение, назначение переменных, передаваемых в модуль и из него, модулей, которые его вызывают, и модулей, которые вызываются из него;
- оператор безусловного перехода или вообще не используется в модуле, или применяется в исключительных случаях только для перехода на выходную точку модуля;
- в тексте модуля необходимо использовать комментарии, в особенности в сложных местах алгоритма;
- имена переменных и модулей должны быть смысловыми, «говорящими»;
- в одной строке не стоит записывать более одного оператора; если для записи оператора требуется больше, чем одна строка, то все следующие операторы записываются с отступами;
- желательно не допускать вложенности более, чем трех уровней;
- следует избегать использования запутанных языковых конструкций и программистских «трюков».

Все эти вместе взятые меры направлены на повышение качества программного обеспечения.

2. Построение графиков и работа с ними



Структура программы

Чтобы проиллюстрировать методику создания больших программ, мы рассмотрим достаточно сложный проект, в котором необходимо выполнить следующие части задания:

- построить на экране оси координат и сделать их разметку
- построить графики заданных вам функций
- найти координаты указанных точек пересечения
- заштриховать замкнутую область, ограниченную кривыми
- вычислить площадь заштрихованной части

Такую работу сложно выполнить сразу, за один день. Разработку большой программы обычно начинают с того, что задача разбивается на несколько более простых подзадач. Для нашей работы такие подзадачи перечислены выше. Будем считать, что каждую их подзадач выполняет специальная процедура. Таким образом, можно сразу определить структуру всей программы и составить основную программу.

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

... // здесь объявляются константы и глобальные переменные

... // здесь записываются все функции и процедуры

//-----
//  Основная программа
//-----
void main ()
{
    int d = DETECT, m;

    initgraph ( &d, &m, "c:\\borlandc\\bgi" );

    Axes();           // построение и разметка осей координат
    PlotF1();         // построение первого графика
    PlotF2();         // построение второго графика
    Cross();          // поиск точек пересечения
    Hatch();          // штриховка
    Area();           // вычисление площади
    getch();          // ждем нажатия на клавишу
    closegraph();     // выключили графический режим
}
```

Таким образом, основная программа состоит только из вызовов процедур. Этих процедур еще нет, мы будем писать их последовательно одну за другой. Чтобы иметь возможность отлаживать программу по частям, прокомментируем строчки с вызовами процедур, поставив в начале каждой строки две наклонные черточки `//`. После того, как очередная процедура составлена и записана (выше основной программы), надо убрать эти символы в соответствующей строчке основной программы, иначе она не будет вызываться.



Оформление программы

Чтобы легче было разбираться в программе, используют специальные приемы оформления. Они не влияют на результат работы программы во время выполнения, но позволяют легче искать ошибки и вносить исправления. Можно выделить следующие принципы.

- Каждая процедура и функция должна иметь заголовок («шапку»), в котором указывают
 - ⇒ название функции
 - ⇒ описание ее работы
 - ⇒ входные и выходные параметры, для функции – возвращаемое значение
 - ⇒ процедуры и функции, которые она вызывает
 - ⇒ процедуры и функции, которые вызывают ее

Простейший пример «шапки» показан выше при описании структуры программы.

- Тело каждого цикла `for`, `while`, `do-while`, включая фигурные скобки, смещается вправо на 2-3 символа относительно заголовка так, чтобы начало и конец цикла были хорошо заметны. Открывающая скобка должна быть на том же уровне, что и закрывающая. Это позволяет сразу обнаруживать непарные скобки.

```
while ( a < b )
{
    ... // тело цикла
}
```

- Аналогичные отступы делаются в условных операторах `if-else` и операторе выбора `switch`.

```
if ( a < b )
{
    ...
}
else
{
    ...
}
```

```
switch ( k )
{
    case 1: ...
    case 2: ...
    default: ...
}
```

- Процедуры и функции должны иметь осмысленные имена, так чтобы можно было сразу вспомнить, что они делают.
- Надо давать переменным осмысленные имена, чтобы их роль была сразу понятна. Так счетчик можно назвать ***count***, переменную, в которой хранится сумма – ***summa*** и т.п.
- Чтобы облегчить чтение программы, полезно использовать пробелы в операторах, например

```
while ( a < b ) { ... }
```

воспринимается легче, чем эта же строка без пробелов.

- Наиболее важные блоки в процедурах и функциях отделяют пустыми строками или строками-разделителями, состоящими, например, из знаков «минус».

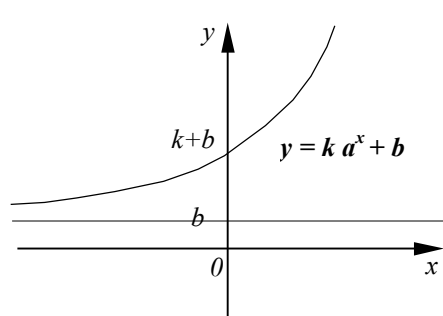
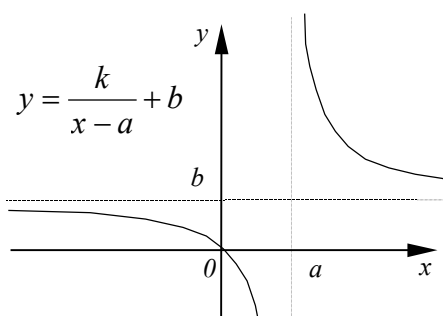
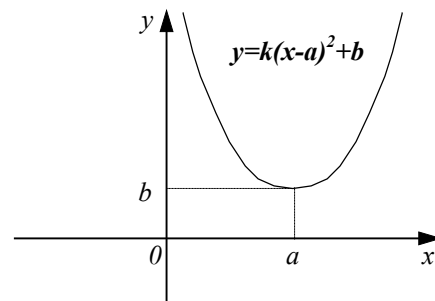
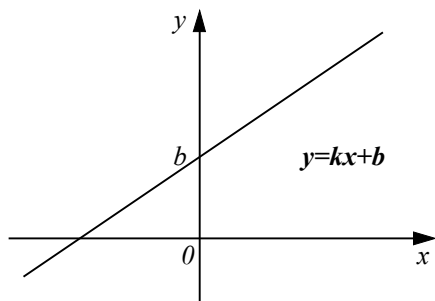


Способы задания функций

В этом разделе рассматриваются различные способы задания кривых на плоскости, используемые в математике.

Функции, заданные в явном виде

Будем считать, что независимой переменной является x , а значением функции — y . В самом простейшем случае известна зависимость $y = f(x)$, то есть зная x можно однозначно определить соответствующее ему значение y . При этом необходимо, чтобы каждому x соответствовало только одно значение y . К числу простейших графиков функций относятся прямая, парабола, гипербола, показательная функция.



Функции, заданные в неявном виде

Не все функции могут быть заданы в явном виде, например те, для которых одному значению x соответствует несколько значений y . В этом случае функцию задают в виде

$$f(x, y) = 0$$

Простейшим примером может служить эллипс или его частный случай - окружность. Уравнение эллипса (которое математики называют **каноническим**) записывается так:

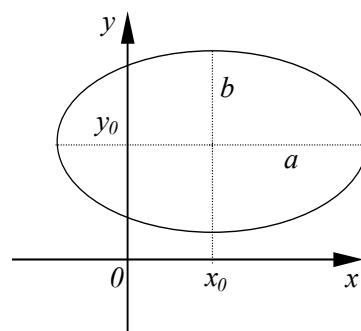
$$\frac{(x - x_0)^2}{a^2} + \frac{(y - y_0)^2}{b^2} = 1,$$

где x_0 и y_0 - координаты центра эллипса, а a и b — длины его полуосей. В частном случае, когда $a=b$, эллипс превращается в окружность.

В простейшем случае можно представить такую кривую как комбинацию из двух или нескольких кривых, каждая из которых задается в явном виде. Например, для окружности можно выразить y в виде

$$y = y_0 \pm b \sqrt{1 - \frac{(x - x_0)^2}{a^2}}$$

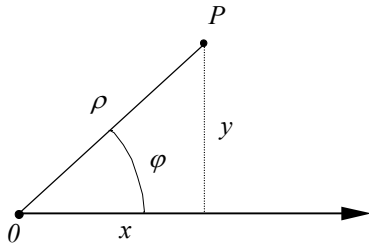
Верхняя часть эллипса соответствует знаку плюс в формуле, а нижняя — знаку минус. При построении эллипсов и окружностей на экране важно проверять область определения.



Функции, заданные в полярных координатах

Для некоторых сложных кривых удастся получить простые выражения, если использовать полярную систему координат, в которой для отсчета используется точка O — **полюс** (начало координат) и **полярная ось** — луч, выходящий из полюса горизонтально вправо — от него

отсчитывается угол. Любая точка P на плоскости имеет две координаты: расстояние от этой точки до полюса ρ и полярный угол между полярной осью и лучом OP . Угол считается положительным при отсчете от полярной оси против часовой стрелки.



Преобразования координат от декартовой системы к полярной и наоборот очевидны:

$$\begin{cases} \rho = \sqrt{x^2 + y^2} \\ \varphi = \arctan \frac{y}{x} \end{cases} \quad \begin{cases} x = \rho \cos \varphi \\ y = \rho \sin \varphi \end{cases}$$

Например, уравнение окружности с центром в начале координат выглядит очень просто в полярной системе координат: $\rho = R$. Как видим, радиус не зависит от угла поворота.

В полярных координатах просто задаются спирали различного типа. например, спираль Архимеда имеет уравнение $\rho = a\varphi$, а логарифмическая спираль задается формулой $\rho = \frac{a}{\varphi}$.

Функции, заданные в параметрическом виде

Для функций, заданных в параметрическом виде, соответствующие друг другу x и y выражаются через третью переменную t , которая называется **параметром**:

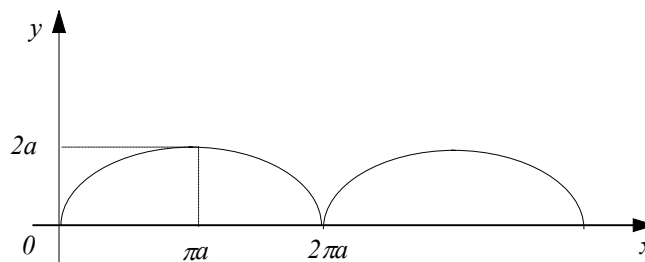
$$\begin{cases} x = f_1(t) \\ y = f_2(t) \end{cases}$$

Например, уравнение эллипса с центром в начале координат записывается в параметрическом виде так

$$\begin{cases} x = a \cos t \\ y = b \sin t \end{cases}$$

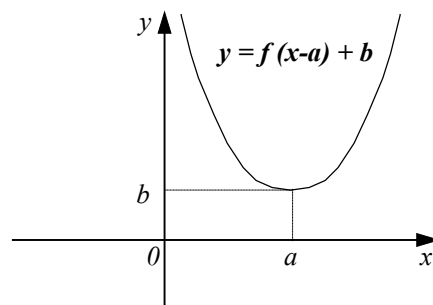
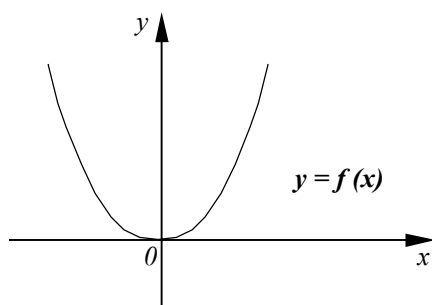
В такой форме очень просто задается **циклоида** — кривая, которую описывает точка окружности, которая катится по прямой. При $\lambda=1$ получаем классическую циклоиду (на рисунке), при $\lambda < 1$ — укороченную, а при $\lambda > 1$ — удлиненную.

$$\begin{cases} x = a(t - \lambda \sin t) \\ y = a(1 - \lambda \cos t) \end{cases}$$

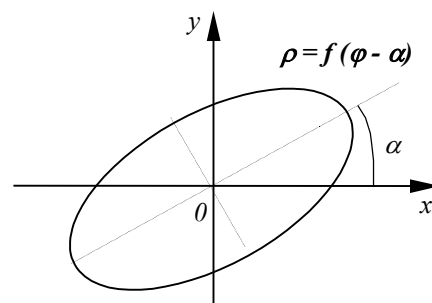
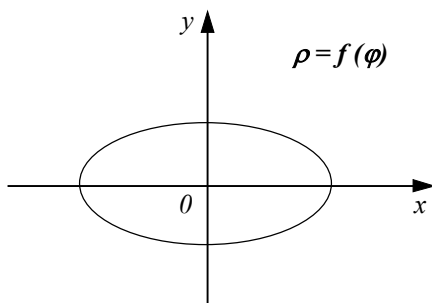


Преобразование координат

Простейшим видом преобразования является **параллельный перенос**. Для того, чтобы сместить график на a по оси x и на b по оси y , надо вычислять значение функции в точках $x-a$ и прибавлять к результату b .



Другим простейшим преобразованием является **поворот** на некоторый угол α . Здесь более удобно использовать полярные координаты. Действительно, при повороте против часовой стрелке надо рассчитать значение $\rho = f(\varphi - \alpha)$ и соответствующие координаты x и y .



Системы координат



Системы координат и обозначения

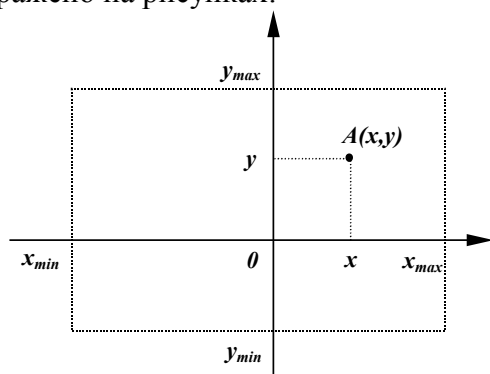
При построении графиков используются две системы координат:

- **математическая** система координат, в которой задана исходная функция
- **экранная** система координат

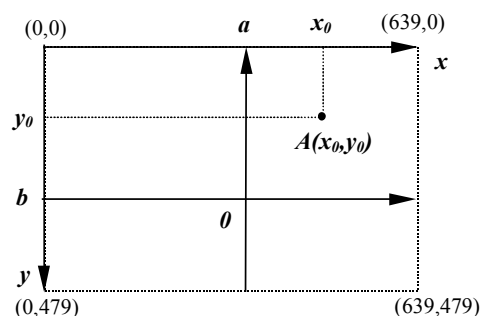
Между ними есть много принципиальных различий. В математической системе плоскость считается бесконечной, на экране мы можем отобразить только заданную прямоугольную часть плоскости. Кроме того, для большинства систем ось x в экранной системе координат направлена вдоль верхней границы экрана влево, а ось y — вдоль левой границы вниз.

Зададим значения x_{min} , x_{max} , y_{min} и y_{max} , ограничивающие прямоугольную часть плоскости, которая будет "проецироваться" на экран монитора. Можно представить себе, что мы смотрим на плоскость в прямоугольное окно.

Будем считать, что экран имеет размеры 640 пикселей в ширину и 480 — в высоту. Для других размеров надо будет просто заменить числа в формулах. Все сказанное выше изображено на рисунках:



математические координаты



экранные координаты

Далее экранные координаты мы будем обозначать индексом "0".

Масштабы и преобразования координат

Для того, чтобы построить на экране оси координат, выберем значения **a** и **b**, показанные на рисунке (координаты точки (0,0) на экране). Для того, чтобы иметь возможность увеличивать и уменьшать график, надо выбрать **масштабный коэффициент k**, который будет показывать, во сколько раз увеличивается каждый отрезок на экране в сравнении с математической системой координат. Фактически **k** – это длина единичного отрезка в экранной системе координат.

Зная **a**, **b** и **k**, можно рассчитать границы видимой области по формулам

$$\begin{aligned}x_{\min} &= -\frac{a}{k}, & x_{\max} &= \frac{(640 - a)}{k}, \\y_{\min} &= -\frac{(640 - b)}{k}, & y_{\max} &= \frac{b}{k},\end{aligned}$$

В программе многие функции будут использовать значения **a**, **b** и **k**, поэтому их лучше сделать глобальными постоянными

```
const a = 320, b = 240, k = 30;
```

Обратите внимание, что все эти константы – целые числа. Если вы хотите использовать дробные значения масштаба, надо написать так:

```
const a = 320, b = 240;
const float k = 3.5;
```

Теперь надо выяснить, как же преобразовать координаты точки **A(x, y)** так, чтобы получить координаты ее образа **(x₀, y₀)** на экране. Каждый отрезок оси **x** растягивается в **k** раз и смещается на **a** относительно левого верхнего угла экрана. Для оси **y** – все почти так же, но надо учесть еще, что ось направлена в **обратном направлении** — вниз. Поэтому в формуле появляется знак минус. Таким образом, формулы для преобразования координат из математической системы в экранную принимают вид:

```
x0 = a + k * x;
y0 = b - k * y;
```

Такой же результат можно получить, если рассмотреть пропорции между соответствующими отрезками на плоскости и на экране.

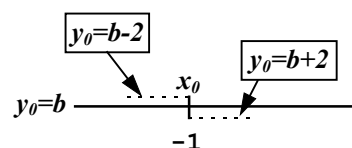
Оси координат

С учетом сказанного выше процедура для рисования осей может выглядеть так:

```
void Axes()
{
    line ( a, 0, a, 479 );
    line ( 0, b, 639, b );
}
```

Теперь осталось добавить в эту процедуру выполнение разметки осей координат — надо нанести отметки и надписи рядом с осями. Покажем, как это выполняется на примере разметки положительной части оси **x** с шагом **1**. Так как длина единичного отрезка на экране равна **k** пикселей, то на положительной части оси **x** будет **(640-a)/k** отметок (не считая нуля). Координата **x₀** (на экране) отметки, соответствующей значению **x=i** (в математической системе) равна **a+i*k**.

Теперь надо сделать отметку на оси и надпись. Часть оси с одной отметкой и надписью показана слева. При известном x_0 надо провести вертикальный отрезок и вывести значение соответствующего ему x .



Если нарисовать отрезок несложно, для вывода надписи придется потрудиться, поскольку в графическом режиме можно выводить на экран только символьные строки, но никак не целые или вещественные числа. Для того, чтобы сделать то, что мы хотим, надо сначала преобразовать число в строку, а затем эту строку вывести на экран. Первая часть выполняется с помощью функции `sprintf`. Она очень напоминает `printf` и `fprintf`, но если первая выводит на экран в текстовом режиме, а вторая — в текстовый файл, то `sprintf` направляет вывод в указанную строку. Конечно, под нее надо заранее выделить место в памяти. В программе это может выглядеть примерно так:

```
int x0;
char s[10];
...
sprintf ( s, "%d", x0 );
```

куда записать результат

формат вывода

Как видим, ее единственная особенность в том, что первым параметром задается строка, в которую записывается результат.

Второй шаг выполняется с помощью функции `outtextxy`, в параметрах которой передаются координаты левого верхнего угла текста и сама символьная строка. В режиме 640×480 один символ занимает ячейку 8×16 пикселей. Будем считать, что надпись занимает два символа, то есть квадрат 16×16 . Чтобы надпись не наезжала на отметку, левый верхнюю границу надписи надо немного сдвинуть вниз.

```
outtextxy ( x0-8, b+4, s );
```

Теперь мы готовы написать всю процедуру построения и разметки осей до конца. Для разметки с шагом 1 надо перебрать все целые значения x между 0 и $(640-a)/k$, и для каждого из них сделать отметку на оси и надпись.

```
void Axes()
{
    int i, x0;
    char s[10];

    line ( a, 0, a, 479 );
    line ( 0, b, 639, b );

    for ( i = 0; i <= (640-a)/k; i ++ )
    {
        x0 = a + k * i;
        line ( x0, b-2, x0, b+2 );
        sprintf ( s, "%d", i );
        outtextxy ( x0-8, b+4, s );
    }

    ... // а здесь надо написать разметку остальных осей
}
```



Построение графиков



Стандартный способ

Построим на экране график функции, заданной в явном виде $y=f_1(x)$, вернее, ту его часть, которая попадает в выбранный прямоугольник. Поскольку функция будет использоваться и в дальнейшем, лучше оформить вычисление y по известному x в виде отдельной функции языка Си, которая принимает единственный вещественный параметр и возвращает вещественное число — значение функции в этой точке. Сама функция может быть сколь угодно сложная, в простейшем случае она выглядит так:

```
float F1 ( float x )
{
    return 1/x;
}
```

Теперь для всех x из интервала $[x_{\min}, x_{\max}]$ надо построить соответствующие точки графика. Однако всего в этом интервале бесконечное множество точек, поэтому на практике придется выбрать конечное множество, например, задать изменение x внутри интервала с шагом h . Каким должен быть шаг?

Поскольку на экране только 640 вертикальных линий, то чаще всего не имеет смысла делать более 640 шагов. Учитывая, что

$$x_{\min} = -\frac{a}{k}, \quad \text{и} \quad x_{\max} = \frac{(640-a)}{k},$$

выбираем

$$h = \frac{x_{\max} - x_{\min}}{640} = \frac{1}{k}.$$

При построении надо проверять область определения функции, чтобы не получить ошибку при делении на ноль, извлечении корня из отрицательного числа и т.п. Для сложных функций лучше оформить свою функцию на языке Си, которая будет возвращать целое число: единицу, если заданное x входит в область определения, и ноль — если не входит. Для нашей функции она может выглядеть так (приводится 2 способа):

```
int ODZF1 ( float x )
{
    if ( x != 0 ) return 1;
    else       return 0;
}
```

```
int ODZF1 ( float x )
{
    return (x != 0);
}
```

Справа записан более профессиональный вариант.

Строить точку на экране надо только тогда, когда она попадает на экран. Если пытаться строить точку, координаты которой находятся вне экрана, мы сотрем какую-то ячейку памяти и результат может быть непредсказуем. Чтобы проверять это, используем специальную процедуру **ShowPoint**, которая будет переводить переданные ей значения координат x и y из математической системы в экранную и, если точка попадает на экран, выводить точку заданного цвета **color**. Эту процедуру удобнее всего разместить выше процедур рисования графиков (PlotF1 и PlotF2), чтобы на момент вызова она была уже известна.

```

void ShowPoint ( float x, float y, int color )
{
    float x0, y0;
    x0 = a + k*x;
    y0 = b - k*y;
    if ( x0 >= 0  &&  x0 < 640  &&
        y0 >= 0  &&  y0 < 480 )
        putpixel(x0, y0, color);
}

```

Теперь, используя приведенные функции, можно составить процедуру, которая рисует график функции на экране:

```

void PlotF1()
{
    float x, h, xmin, xmax;
    h = 1 / k;
    xmin = - a / k;
    xmax = (640 - a) / k;
    for ( x = xmin; x <= xmax; x += h )
        if ( ODZF1(x) )
            ShowPoint (x, F1(x), RED);
}

```

Если надо построить несколько графиков, это можно сделать в одном цикле, вызывая каждый раз соответствующие функции для вычисления значения y и проверки области определения.



Функции, заданные в полярных координатах

Если функция задана в полярных координатах, независимой переменной в цикле будет не координата x , а угол поворота радиус-вектора φ (угол между горизонтальной осью и вектором из начала координат в данную точку). Надо иметь в виду, что угол измеряется в радианах. Угол 90 градусов равен $\pi/2$ радиан (в языке Си константа π обозначается **M_PI**). Диапазон углов от 0 до 2π радиан (360 градусов) соответствует одному обороту вокруг начала координат.

Если требуется, также надо использовать область определения функции (чтобы избежать деления на нуль и извлечения корня из отрицательного числа). Шаг изменения угла подбирается опытным путем так, чтобы не было видно, что линия состоит из отдельных точек.

```

void PlotPolar ()
{
    float phi, phiMin = 0., phiMax = 2*M_PI,
        h = 0.001, x, y, ro;
    for ( phi = phiMin; phi <= phiMax; phi += h )
        if ( ODZF1(phi) ) {
            ro = F1(phi);
            x = ro * cos(phi);
            y = ro * sin(phi);
            ShowPoint (x, y, RED);
        }
}

```

Функции, заданные в параметрическом виде

Если функция задана в параметрическом виде, независимой переменной в цикле будет параметр t . Диапазон его изменения (значения переменных t_{Min} и t_{Max} в программе) надо подбирать по справочнику или экспериментально. Шаг изменения параметра t также подбирается опытным путем. Для каждого значения t вычисляются x и y (предположим, что это делают функции $F_x(t)$ и $F_y(t)$). Общий порядок действия аналогичен предыдущим случаям.

```
void PlotParameter ()
{
    float  t, tMin = -10., tMax = 10.,
           h = 0.001, x, y;
    for ( t = tMin; t <= tMax; t += h )
        if ( ODZF1(t) ) {
            x = Fx(t);
            y = Fy(t);
            ShowPoint (x, y, RED);
        }
}
```

Как найти точки пересечения?

На следующем этапе работы надо найти точки пересечения графиков. Если даны две функции $f_1(x)$ и $f_2(x)$, координаты точек их пересечения по оси x удовлетворяют уравнению

$$f_1(x) = f_2(x)$$

К сожалению, точно решить это уравнение аналитически можно только в некоторых простейших случаях. Если функции нелинейные, чаще всего аналитического решения не существует. При этом задачу решать надо, и используются **численные методы**, которые всегда являются **приближенными**. Это означает, что мы всегда получаем решение с некоторой **ошибкой**, хотя она может быть очень и очень маленькой (даже одна миллионная и меньше).

Метод перебора

Примитивный метод решения уравнения заключается в следующем. Пусть мы знаем, что на интервале $[a, b]$ графики имеют одну и только одну точку пересечения и требуется найти координату точки пересечения x^* с точность ε .

Разобьем интервал $[a, b]$ на кусочки длиной $h=\varepsilon$ и найдем из них такой отрезок $[x_1^*, x_2^*]$, на котором уравнение имеет решение, то есть линии пересекаются. Для этого перебираем все возможные x от a до b и проверяем для каждого значение произведения

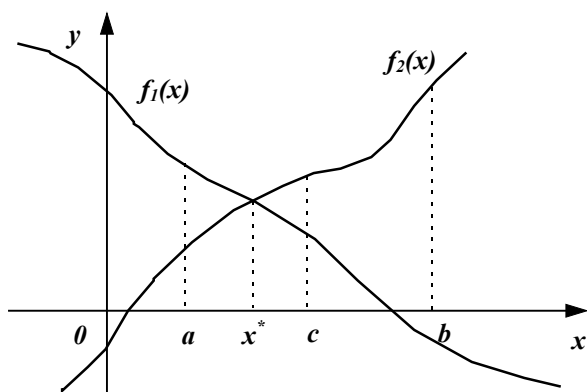
$$g_1 = f_1(x) - f_2(x), \quad g_2 = f_1(x+\varepsilon) - f_2(x+\varepsilon).$$

Если функции g_1 и g_2 имеют разные знаки, значит на интервале $[x, x+\varepsilon]$ есть решение уравнения.

Можно поступать: найти такое x , для которого величина функции g_1 (то есть разность двух исходных функций) минимальна, и принять ее за x^* .

Метод деления отрезка пополам

Один из самых простых численных методов решения уравнений — **метод деления пополам** или **дихотомия**. Он связан со старинной задачей о поиске льва в пустыне. Надо разделить пустыню на 2 равные части и определить, в которой из них находится лев. Затем эту половину снова делим на две части и т.д.. В конце концов делить станет нечего — лев вот он!



проходит выше, чем $f_2(x)$, то в точке c — наоборот. Математически это можно выразить условием

$$(f_1(a) - f_2(a))(f_1(c) - f_2(c)) < 0$$

Если условие верно, то на участке $[a, c]$ есть пересечение.

Когда же закончить деление отрезка пополам? Видимо тогда, когда мы достигнем заданной точности ε , то есть когда ширина отрезка $[a, b]$ станет меньше ε . Функция для вычисления корня с точностью **eps** приводится ниже:

```
float Root( float A, float B, float eps )
{
    float C, gA, gC;
    while ( fabs(B-A) > eps ) {
        C = ( A + B ) / 2.;
        gA = F1(A) - F2(A);
        gC = F1(C) - F2(C);
        if ( gA*gC < 0 ) B = C;
        else          A = C;
    }
    return ( A + B ) / 2.;
}
```

Она возвращает приближенное значение координаты x точки пересечения. У этой функции есть один недостаток — она жестко завязана на функции $f_1(x)$ и $f_2(x)$. Хорошо бы сделать так, чтобы в параметрах этой функции можно было передать **адреса функций**, которые надо вызывать.

Этого можно добиться, если объявить новый тип данных — **указатель на функцию**. Единственное ограничение — все функции, которые передаются в процедуру, должны быть одного типа, то есть принимать одинаковые аргументы (в нашем случае — `float x`) и возвращать значения одинаковых типов (в нашем случае — `float`). Объявить новый тип (назовем его **func**) можно в начале программы с помощью директивы `typedef`

```
typedef float (*func)( float x );
```

Это достаточно сложное определение говорит о том, что теперь введен новый тип данных — указатель на функцию, которая принимает один вещественный аргумент и возвращает вещественный результат, и тип этот называется **func**. Тогда функция **Root** может выглядеть так:

Пусть мы знаем, что на интервале $[a, b]$ графики имеют одну и только одну точку пересечения (это важно!). Идея метода такова: найдем середину отрезка $[a, b]$ и назовем ее c . Теперь проверяем, есть ли пересечение на участке $[a, c]$. Если есть, ищем дальше на этом интервале, а если нет, то на интервале $[c, b]$.

Заметим, что если графики пересекаются на отрезке $[a, c]$, то разность этих функций меняет знак — если в точке a функция $f_1(x)$

```

float Root( func f1, func f2,
            float A, float B, float eps )
{
    ...
    gA = f1(A) - f2(A);
    gC = f1(C) - f2(C);
    ...
}

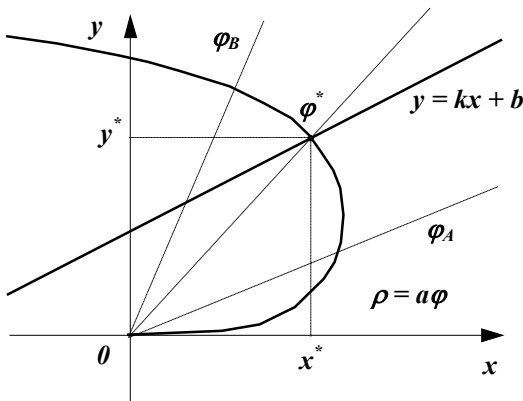
```

Все остальные строки не меняются. При вызове в первых двух параметрах надо указать адреса (то есть имена) функций, которые надо использовать, например, так

```
x1 = Root ( F1, F2, -1., 2., 0.0001 );
```

Функции, заданные в полярных координатах

Возможна такая ситуация, когда одна из кривых задана в полярных координатах, а вторая — в явном виде. В этом случае задача определения координат точек пересечения становится несколько сложнее.



Покажем принцип решения задачи на примере. Пусть требуется найти пересечение спирали $\rho = a\varphi$ и прямой $y = kx + b$. Мы будем искать точку пересечения методом деления отрезка пополам. Сначала построим оба графика и определим углы φ_A и φ_B , между которыми находится точка пересечения графиков. В данном случае можно принять $\varphi_A = 0$ и $\varphi_B = \pi/2$. Наша задача — определить угол φ^* и соответствующие ему x^* и y^* .

Заметим, что при $\varphi < \varphi^*$ спираль проходит ниже прямой, поэтому если мы рассчитаем для некоторого φ значение ρ и соответствующие ему x_ρ и y_ρ , а также ординату прямой $y_1 = kx_\rho + b$, то выполнится условие $y_\rho < y_1$ или $y_\rho - y_1 < 0$. Аналогично при $\varphi > \varphi^*$ спираль проходит выше прямой, поэтому выполнится условие $y_\rho > y_1$ или $y_\rho - y_1 > 0$.

Таким образом можно применить метод отрезка деления пополам, поскольку разность $y_\rho - y_1$ меняет знак при переходе φ через φ^* . Общую идею можно записать в виде следующего алгоритма:

- определить интервал углов $[\varphi_A, \varphi_B]$, в котором находится угол φ^* соответствующий точке пересечения
- найти середину интервала — угол $\varphi_C = (\varphi_A + \varphi_B)/2$
- вычислить $\rho(\varphi_A)$ и соответствующие $y_\rho(\varphi_A)$ и $y_1(\varphi_A)$
- вычислить $\rho(\varphi_C)$ и соответствующие $y_\rho(\varphi_C)$ и $y_1(\varphi_C)$
- если значения $y_\rho(\varphi_A) - y_1(\varphi_A)$ и $y_\rho(\varphi_C) - y_1(\varphi_C)$ имеют разные знаки (есть пересечение), повторить поиск на интервале $[\varphi_A, \varphi_C]$, иначе искать на $[\varphi_C, \varphi_B]$
- поиск заканчивается, когда длина интервала $[\varphi_A, \varphi_B]$ будет меньше заданной точности

Для вычисления x и y по заданному углу φ для функции, заданной в полярных координатах, удобно использовать процедуру. Она должна вернуть в виде результата два значения, которые

можно передать через **изменяемые параметры** (перед именем которых в заголовке функции стоит знак **&**):

```
void Spiral( float phi, float &x, float &y )
{
    float rho, a = 1;
    rho = a * phi;
    x = rho * cos(phi);
    y = rho * sin(phi);
}
```

Эту процедуру надо вызывать два раза — один раз для угла φ_A , второй раз — для φ_C . Полностью функция, вычисляющая координаты **x** и **y** точки пересечения приведена ниже. Точность **eps** определяется как длина диапазона **углов** $[\varphi_A, \varphi_B]$, при котором процесс прекращается.

```
void Root1( float phiA, float phiB,
            float eps, float &x, float &y )
{
    float phiC, xA, yA, xC, yC, gA, gC;
    while ( fabs(phiB-phiA) > eps ) {
        phiC = (phiA + phiB) / 2.;
        Spiral ( phiA, xA, yA );
        Spiral ( phiC, xC, yC );
        gA = yA - F1(xA);
        gC = yC - F1(xC);
        if ( gA*gC < 0 ) phiB = phiC;
        else             phiA = phiC;
    }
    x = xA;
    y = yA;
}
```

Аналогичным способом находятся точки пересечения кривых, одна из которых задана в параметрическом виде, а другая — в явном.



Функции, заданные в параметрическом виде

Пусть теперь одна кривая задана в параметрической форме, а вторая - в явном виде. Независимой переменной является параметр **t**, поэтому, построив графики, надо определить интервал значений параметра $[t_A, t_B]$, внутри которого находится корень уравнения. Далее корень находим методом деления пополам **диапазона изменения параметра**. Пусть функции $F_x(t)$ и $F_y(t)$ вычисляют значения координат параметрической кривой для заданного **t**. Тогда алгоритм выглядит так:

- найти t_C — середину интервала $[t_A, t_B]$
- определить координат кривой (x_A, y_A) и (x_C, y_C) для значений параметра t_A и t_C
- сравнить значения y_A и y_C со значениями второй функции в точках x_A и x_C , если разности

$$f_2(x_A) - y_A \quad \text{и} \quad f_2(x_C) - y_C$$

имеют разный знак, то пересечение есть на интервале $[t_A, t_C]$, если они одного знака, тогда пересечение есть на интервале $[t_C, t_B]$

```

void Root2( float tA, float tB,
            float eps, float &x, float &y )
{
    float tC, xA, yA, xC, yC, gA, gC;

    while ( fabs(tB-tA) > eps ) {
        tC = (tA + tB) / 2.;
        xA = Fx(tA); yA = Fy(tA);
        xC = Fx(tC); yC = Fy(tC);
        gA = yA - F2(xA);
        gC = yC - F2(xC);
        if ( gA*gC < 0 ) tB = tC;
        else             tA = tC;
    }

    x = xA;
    y = yA;
}

```



Общий случай

Для общего случая (когда, например, обе кривые заданы в полярных координатах) можно использовать следующий подход:

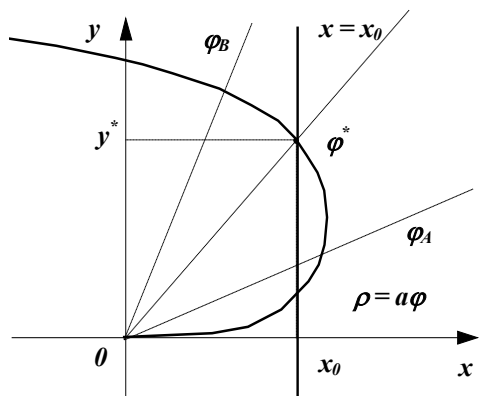
- выбрать независимую переменную, по которой будет проводиться поиск решения
- найти функцию, которая меняет знак в точке пересечения кривых (это самый сложный этап, поскольку все зависит от конкретного вида и формы задания этих кривых)
- найти интервал $[A, B]$ изменения независимой переменной, такой, что внутри него содержится точка пересечения и выбранная функция не меняет знак нигде, кроме точки пересечения
- применить один из численных методов поиска решения в заданном интервале, например, метод деления отрезка пополам



Пересечение с вертикальной прямой

Предложенный в предыдущем параграфе алгоритм можно применить и для случая, показанного на рисунке, когда одна из кривых — вертикальная прямая, а вторая задана полярных координатах.

- выберем φ в качестве независимой переменной
- функция, меняющая знак в точке пересечения: $x_\rho(\varphi) - x_0$
- определяем по графику интервал $[\varphi_A, \varphi_B]$
- применяем метод деления пополам



Единственная сложность заключается в проверке наличия корня в некотором интервале изменения угла $[\varphi_A, \varphi_C]$. Для этого поступают следующим образом:

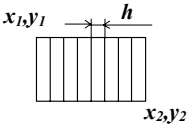
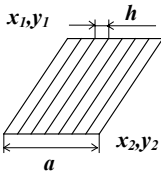
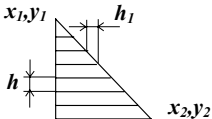
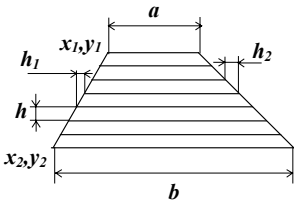
- рассчитывают координаты x_A и x_C точек кривой, соответствующих углам φ_A и φ_C
- если разности $x - x_A$ и $x - x_C$ имеют разный знак, то пересечение есть на интервале $[\varphi_A, \varphi_C]$, если они одного знака, тогда пересечение есть на интервале $[\varphi_C, \varphi_B]$

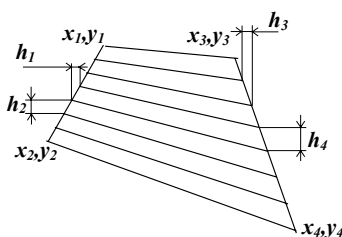
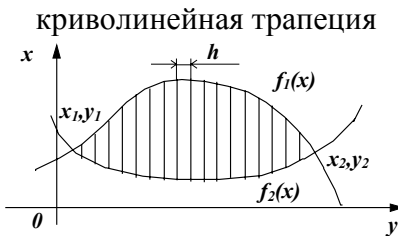
Для кривых, заданных в параметрическом виде, точки пересечения с вертикальной прямой вычисляются почти так же. В этом случае независимой переменной является параметр и координаты x_A и x_C рассчитываются с помощью функции $F_x(t)$.



Штриховка замкнутой области

В данном разделе вы научитесь делать штриховку простейших геометрических фигур на плоскости. Везде предполагается, что надо сделать ровно N линий штриховки (не считая границ фигуры). Для большинства программ потребуется **шаг штриховки** — расстояние между линиями. Если на интервале длиной L надо провести N линий, не считая границ, то шаг будет равен $L/(N+1)$, так как интервалов на 1 больше, чем линий. Этот принцип используется везде далее.

Фигура	Метод штриховки и программа
<p>прямоугольник</p> 	<p>В цикле меняем x от x_1 до x_2 с шагом h, линии строго вертикальные:</p> <pre>for (x = x1; x <= x2; x += h) line (x, y1, x, y2);</pre>
<p>параллелограмм</p> 	<p>Линии наклонные, параллельные, верхний конец отрезка смещен на a вправо относительно нижнего:</p> <pre>for(x = x1; x <= x1+a; x += h) line (x, y1, x-a, y2);</pre>
<p>треугольник</p> 	<p>Единственная сложность заключается в том, что для каждой следующей линии координата x конца отрезка смещается на $h_1 = \frac{x_2 - x_1}{N + 1}$ по отношению к предыдущей, одновременно с координатой y:</p> <pre>h1 = (x2 - x1) / (N + 1); xe = x1; for(y = y1; y <= y2; xe += h1, y += h) line (x1, y, xe, y);</pre>
<p>трапеция</p> 	<p>Сразу меняются координаты x для обоих концов отрезка:</p> <pre>h1 = (x1 - x2) / (N + 1); h2 = (b - a - x1 + x2) / (N + 1); xs = x1; xe = x1 + a; for(y = y1; y <= y2; xs -= h1, xe += h2, y += h) line (xs, y, xe, y);</pre>

<p>два отрезка</p> 	<p>Надо сделать штриховку так, чтобы каждый отрезок был разделен на $N+1$ равных отрезков: все координаты меняются синхронно на величины соответствующих шагов</p> <pre> h1 = (x1 - x2) / (N + 1); h2 = (y2 - y1) / (N + 1); h3 = (x4 - x3) / (N + 1); h4 = (y4 - y3) / (N + 1); xs = x1; xe = x3; ye = y3; for(y = y1; y <= y2; xs -= h1, xe += h3, y += h2, ye += h4) line (xs, y, xe, ye); </pre>
<p>криволинейная трапеция</p> 	<p>Обычно функции $f_1(x)$ и $f_2(x)$ заданы в математической системе координат, поэтому для рисования отрезка его координаты надо преобразовать к экранным:</p> <pre> for(x = x1; x <= x2; x += h) { x0 = a + kx*x; y01 = b - ky*f1(x); y02 = b - ky*f2(x); line (x0, y01, x0, y02); } </pre> <p>Функции $f_1(x)$ и $f_2(x)$, ограничивающие границы фигуры, иногда состоят из нескольких частей, которые имеют разные уравнения. В этом случае для их вычисления можно использовать отдельную функцию языка Си.</p>



Площадь замкнутой области



Общий подход

Последняя часть работы — вычисление площади заштрихованной фигуры. Существует довольно много разных методов решения этой задачи. Мы рассмотрим четыре самых простых метода и попытаемся сравнить их. Все они являются численными и позволяют найти значение площади только приближенно. Тем не менее, на практике почти всегда можно найти площадь с требуемой точностью (за счет увеличения времени вычислений).

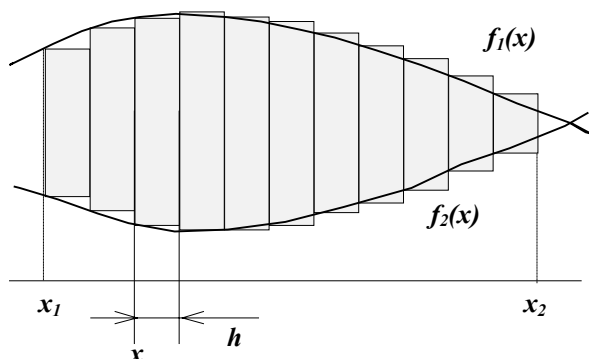
Фактически площадь равна определенному интегралу и часто может быть вычислена аналитически, но эта тема изучается только в конце школьного курса и в высшей математике

Представим себе, что фигура состоит из нескольких простых фигур, для которых мы можем легко подсчитать площадь. В этом случае надо вычислить площади всех этих фигур и сложить их. Но у нашей фигуры границы — кривые линии, поэтому представить ее точно как сумму многоугольников невозможно. Таким образом, используя такой подход, мы вычисляем площадь с некоторой ошибкой, но эту ошибку можно сделать достаточно малой (например, 0.00001).

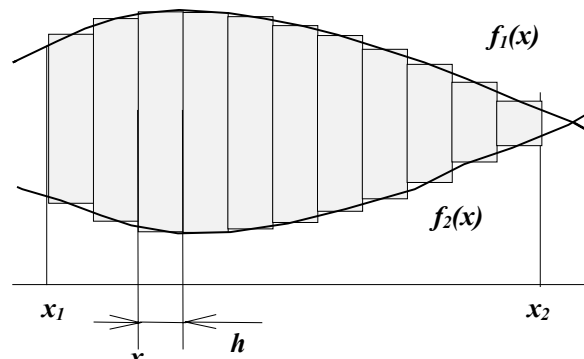


Методы прямоугольников

Простейшими методами вычисления площади являются методы прямоугольников. Из названия ясно, что фигура разбивается на прямоугольники, например так, как показано на рисунках. Обычно ширина h всех прямоугольников выбирается одинаковой, хотя в принципе это не обязательно.



Метод левых прямоугольников



Метод средних прямоугольников

Рассмотрим прямоугольник, левая граница которого имеет координату x . Его высоту можно определить разными способами. Наиболее часто используются три способа:

- высота равна $f_1(x) - f_2(x)$ — метод **левых** прямоугольников
- высота равна $f_1(x+h) - f_2(x+h)$ — метод **правых** прямоугольников
- высота равна $f_1(x+0.5h) - f_2(x+0.5h)$ — метод **средних** прямоугольников

Можно доказать, что самый точный из них — метод средних прямоугольников, он дает наименьшую ошибку.

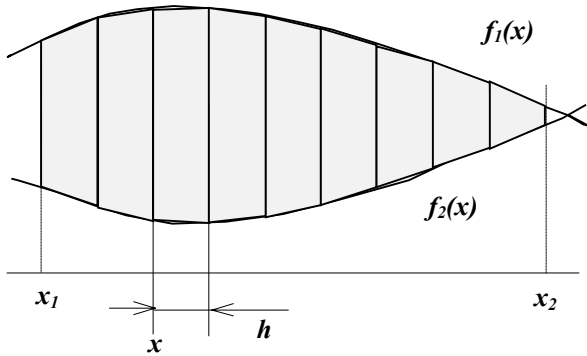
Для того, чтобы вычислить площадь всей фигуры, надо сложить площади всех прямоугольников, имеющих левые границы с координатами от x_1 до x_2-h . Поскольку ширина всех прямоугольников выбирается равной h , можно сложить в цикле все высоты, а затем сумму умножить на h — это будет выполняться быстрее. Ниже приведена программа, вычисляющая площадь S методом средних прямоугольников.

```
float S = 0., x, h = 0.001;
for ( x = x1; x <= x2-h; x += h )
    S += f1(x+h/2) - f2(x+h/2);
S *= h;
```

Если надо повысить точность вычислений, уменьшают шаг h . Методы **левых** и **правых** **прямоугольников** имеют погрешность порядка h , что означает, что при уменьшении h в два раза погрешность уменьшается тоже в 2 раза. Метод **средних** **прямоугольников** имеет погрешность порядка h^2 , то есть при уменьшении h в два раза погрешность уменьшается в 4 раза.

Метод трапеций

Среди простых фигур, которыми можно приближенно заменять "полосы" криволинейной фигуры, удобно использовать **трапеции**, для которых площадь определяется без труда.



Метод трапеций

Площадь одной трапеции равна произведению полусуммы оснований, на высоту. Высота у всех одинакова и равна h , а основания для трапеции, имеющей координату левой границы x , равны, соответственно,

$$f_1(x) - f_2(x) \quad \text{и} \quad f_1(x+h) - f_2(x+h)$$

Заметим также, что правое основание одной трапеции равно левому основанию следующей, что можно учесть при суммировании.

Для вычисления общей площади надо сложить площади всех трапеций, или, что равносильно, сложить их основания и умножить на $h/2$. Каждое основание входит в сумму два раза (кроме левой границы первой трапеции и правой границы последней). Поэтому программа может выглядеть так:

```
float S, x, h = 0.001;
S = (f1(x1) - f2(x1) + f1(x2) - f2(x2)) / 2.;
for ( x = x1+h; x <= x2-h; x += h )
    S += f1(x) - f2(x);
S *= h;
```

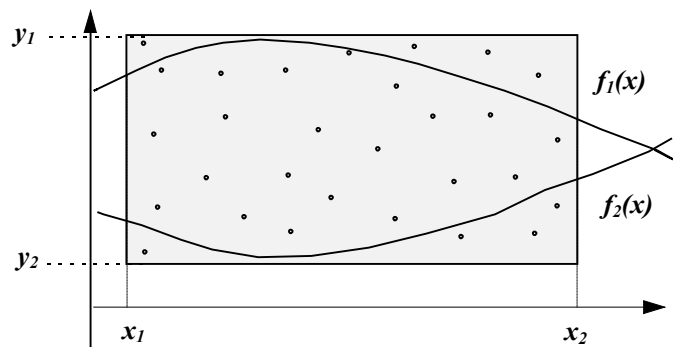
Метод трапеций имеет порядок h^2 , так же, как и метод средних прямоугольников. Удивительно, но его точность ровно в два раза хуже, чем для метода средних прямоугольников.

Метод Монте-Карло

Монте-Карло — всемирный центр игорного бизнеса — дал название большой группе методов, которые позволяют приближенно решать сложные задачи моделирования с помощью случайных чисел. В частности, мы рассмотрим применение метода Монте-Карло (метода статистических испытаний) для вычисления площади фигуры.

Идея метода очень проста. Сначала вся фигура, площадь которой надо определить, заключается внутрь другой фигуры, для которой площадь легко вычисляется. Чаще всего этой фигурой-контейнером является прямоугольник, хотя можно использовать окружность и другие фигуры.

Далее с помощью датчика случайных чисел с **равномерным распределением** генерируются случайные координаты (x, y) точки **внутри** прямоугольника и определяют, попала точка на фигуру или нет. Такие испытания проводятся N раз (для увеличения точности N должно быть большим, порядка нескольких тысяч или даже нескольких миллионов).



Метод Монте-Карло

Пусть из N точек, полученных таким образом, M попали на фигуру. Тогда, считая, что распределение точек в прямоугольнике **равномерное**, площадь фигуры вычисляют как

$$S = \frac{M}{N}(x_2 - x_1)(y_2 - y_1)$$

где произведение двух множителей в скобках представляет собой площадь прямоугольника-контейнера.

```
long i, N = 10000, M = 0;
float S, x, y;
for ( i = 1; i <= N; i ++ ) {
    x = RandFloat (x1, x2);
    y = RandFloat (y1, y2);
    if ( InsideFigure(x,y) ) M ++;
}
S = M * (x2 - x1) * (y2 - y1) / N;
```

Приведенная программа использует функцию RandFloat, которая возвращает вещественные числа, равномерно распределенные в заданном интервале:

```
float RandFloat( float min, float max )
{
    return (float)rand()*(max - min) / RAND_MAX + min;
}
```

Другая функция — InsideFigure — возвращает 1, если точка (x, y) попала внутрь фигуры, и 0, если нет:

```
int InsideFigure( float x, float y )
{
    return f1(x) >= y && y >= f2(x);
}
```

Метод Монте-Карло дает не очень точные результаты. Его точность сильно зависит от равномерности распределения случайных точек в прямоугольнике.

Конечно, в реальных условиях этот метод следует использовать только тогда, когда других способов решения задачи нет или они чрезвычайно трудоемки. Рассмотрим задачу, для которой наиболее приемлем метод Монте-Карло:

Задача 1. N треугольников заданы координатами своих вершин. Требуется найти площадь фигуры, образованной объединением всех треугольников.

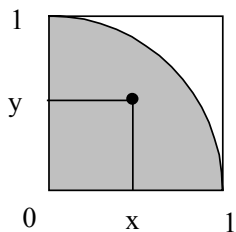
Сложность. Треугольники могут пересекаться, поэтому фигура может состоять из нескольких частей и иметь границу сложной формы.

Решение. Решение состоит из двух этапов:

1. Перебором определяем прямоугольник, в котором находятся все заданные треугольники, то есть, находим минимальные и максимальные координаты вершин по каждой оси.
2. Используем метод Монте-Карло, где функция InsideFigure возвращает 1 в том случае, если точка попала хотя бы в один треугольник.

Задача 2. Используя метод Монте-Карло, вычислить приближенно число π .

Решение. Известно, что площадь круга равна $S = \pi R^2$, где R – радиус круга. Тогда $\pi = S / R^2$, то есть, зная радиус окружности и определив численно ее площадь, можно приближенно рассчитать значение числа π .



Выберем окружность единичного радиуса с центром в начале координат и рассмотрим ее четверть. Для использования метода Монте-Карло нам надо, используя датчик случайных чисел, равномерно заполнить точками (пусть их общее количество будет N) квадрат со стороной 1 и сосчитать, сколько из них окажется внутри окружности (это число обозначим через M). Тогда площадь всей окружности равна $4M/N$, а так как ее радиус равен 1, это же число примерно равно π .

Таким образом, для решения этой задачи надо N раз выполнить две операции:

1. получить случайные координаты (x, y) точки внутри квадрата — два случайных числа в интервале от 0 до 1,
2. определить, находится ли точка (x, y) внутри окружности, то есть, выполняется ли условие $x^2 + y^2 \leq R^2 = 1$.

3. Вычислительные методы



Целочисленные алгоритмы

К этой группе относятся некоторые классические алгоритмы, которые были известны еще древним математикам.



Алгоритм Евклида (I)

Алгоритм Евклида (3 в. до н.э.) служит для вычисления наибольшего общего делителя (НОД) двух натуральных чисел. Оригинальный алгоритм Евклида основан на равенстве

$$\text{НОД}(a, b) = \text{НОД}(a-b, b) = \text{НОД}(a, b-a).$$

Чтобы не работать с отрицательными числами, можно сформулировать алгоритм так:

Алгоритм Евклида. Заменяем большее из двух чисел разностью большего и меньшего до тех пор, пока они не станут равны — это и есть НОД.

Функция для вычисления НОД может быть записана в двух вариантах:

рекурсивная

```
int NOD ( int a, int b )
{
    if ( a == b ) return a;
    if ( a < b )
        return NOD(a, b-a);
    else return NOD(a-b, b);
}
```

нерекурсивная

```
int NOD ( int a, int b )
{
    while ( a != b ) {
        if ( a > b ) a -= b;
        else      b -= a;
    }
    return a;
}
```

Конечно, в данном случае нерекурсивная форма лучше: она работает быстрее и не расходует стек попусту.



Алгоритм Евклида (II)

Первый вариант алгоритма Евклида медленно работает в том случае, если числа сильно отличаются (например, вычисление **НОД(2, 1998)** потребует 998 шагов цикла). Поэтому чаще используют *модифицированный алгоритм Евклида*, основанный на использовании равенства

$$\text{НОД}(a, b) = \text{НОД}(a, b \% a) = \text{НОД}(a \% b, b)$$

Модифицированный алгоритм Евклида. Заменяем большее из двух чисел остатком от деления большего на меньшего до тех пор, пока меньшее не станет равно нулю. Тогда второе число и есть НОД.

Запишем на языке Си только нерекурсивный вариант:

```
int NOD ( int a, int b )
{
    while ((a != 0) && (b != 0)) {
        if ( a > b ) a = a % b;
        else      b = b % a;
    }
    return a+b;
}
```



Наименьшее общее кратное

Наименьшим общим кратным (НОК) двух чисел называется наименьшее число, которое делится без остатка на оба исходных числа.

Найти наименьшее общее кратное можно очень просто, если знать НОД исходных чисел. Действительно, пусть

$$x = x_1 \cdot d \quad \text{и} \quad y = y_1 \cdot d$$

где $d = \text{НОД}(x, y)$. Тогда

$$\text{НОК}(x, y) = xy_1 = x_1y = \frac{xy}{\text{НОД}(x, y)}$$

Для вычисления НОД можно использовать алгоритм Евклида.



Решето Эратосфена

Другая задача, которую решили математики Древней Греции — нахождение всех простых чисел в интервале от 1 до заданного числа N . Самым быстрым и поныне считается *алгоритм Эратосфена* (275-195 гг. до н.э.). Существует предание, что Эратосфен выписал в ряд на папирусе все натуральные числа и затем прокалывал каждое второе (то есть делящееся на 2), потом — каждое третье, и т.д. В конце этой процедуры остаются "непроколотыми" только простые числа, которые делятся только на себя и на 1.

Вместо папируса мы будем использовать массив A , в котором элемент $A[i]$ для всех i от 1 до N принимает два возможных значения:

1. Единица, число "не проколото" и является кандидатом на простые.
2. Нуль, число "проколото" и больше не рассматривается.

В целях экономии памяти можно выбрать не целые переменные (занимающие не менее 2 байт), а символьные (например, `unsigned char`, которые занимают 1 байт). Кроме того, для экономии памяти еще в 8 раз можно рассматривать отдельные биты числа, но этот вариант мы не затрагиваем.

Будем перебирать все числа k от 2 до \sqrt{N} (включительно) и "вычеркивать" числа $2k, 3k, \dots$ в цикле. Очевидно, что надо использовать только те числа, которые еще не вычеркнуты. Ниже полностью приведена программа, реализующая этот алгоритм. Она имеет одно ограничение: $N < 65535$. Это связано с тем, что в данной версии языка Си невозможно просто выделить блок памяти больше 64 Кб. Заметьте также, что надо выделить место в памяти под $N+1$ элемент, чтобы использовать элементы с $A[1]$ по $A[N]$. При этом элемент $A[0]$ не используется. Можно этого избежать, но это усложнит программу.

```

#include <stdio.h>
void main()
{
    unsigned char *A;
    long int i, k, N;

    printf ("Введите максимальное число ");
    scanf ( "%ld", &N );

    A = new unsigned char[N+1];
    if ( A == NULL ) return;

    for ( i = 1; i <= N; i ++ ) A[i] = 1;
        for ( k = 2; k*k <= N; k ++ )
            if ( A[k] != 0 )
                for ( i = k+k; i <= N; i += k ) A[i] = 0;

    for ( i = 1; i <= N; i ++ )
        if ( A[i] == 1 ) printf ( "%ld\n", i );
}

```

← выделить память под массив

Главное **преимущество** этого алгоритма — высокая скорость работы, основной **недостаток** — большой объем необходимой памяти. Он демонстрирует одну из принципиальных проблем программирования: как правило, повышение быстродействия алгоритма требует увеличения необходимого объема памяти, а экономия памяти приводит к снижению быстродействия. В реальных ситуациях чаще всего приходится идти на компромисс.



Многоразрядные целые числа

Как работать в языке Си с многоразрядными целыми числами, которые не помещаются в одну ячейку памяти, даже типа `long int`? Очевидно, что надо отвести на них несколько ячеек, но при этом придется вручную реализовывать все операции с таким числом. Существует два метода

1. Число хранится в виде символьного массива, в котором каждый элемент обозначает одну цифру от 0 до 9.
2. Число хранится в виде массива целых чисел (`int` или `long int`), где каждый элемент обозначает одну или несколько цифр.

Мы рассмотрим второй способ. Для экономии места в памяти выгоднее размещать в одной ячейке как можно больше цифр. При этом надо следить, чтобы результаты всех промежуточных операций не превышали максимального значения для выбранного типа данных.

Рассмотрим, например, число 12 345678 901234 567890. Его можно записать в виде

$$12 \cdot (10^6)^3 + 345678 \cdot (10^6)^2 + 901234 \cdot (10^6)^1 + 567890 \cdot (10^6)^0$$

что соответствует записи в системе счисления с основанием **$d=1000000$** . Цифрами в этой системе служат многоразрядные десятичные числа. Для записи этого конкретного числа нам понадобится всего 4 ячейки типа `long int`.

Теперь надо научиться выполнять арифметические операции с такими числами. Рассмотрим в общем виде умножение числа

$$A = a_n d^n + a_{n-1} d^{n-1} + \dots + a_2 d^2 + a_1 d + a_0$$

на некоторое число **B** . В результате получим третье число **C** , которое также может быть представлено в системе счисления с основанием **d** :

$$C = c_m d^m + c_{m-1} d^{m-1} + \dots + c_2 d^2 + c_1 d + c_0$$

Очевидно, что $c_0 = a_0 B \% d$, при этом в следующий разряд добавляется перенос $r_1 = a_0 B / d$, где остаток от деления отбрасывается. Для следующих разрядов можно составить таблицу

$$\begin{array}{ll} c_0 = a_0 B \% d & r_1 = a_0 B / d \\ c_1 = (a_1 B + r_1) \% d & r_2 = (a_1 B + r_1) / d \\ c_2 = (a_2 B + r_2) \% d & r_3 = (a_2 B + r_2) / d \\ \dots & \dots \end{array}$$

Вычисления заканчиваются, когда одновременно выполняются два условия:

1. Все цифры числа A обработаны.
2. Перенос в следующий разряд равен нулю.

Для удобства фактическая длина числа обычно хранится в отдельной переменной.

Вычисление 100!

Вычислим значение

$$100! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot 99 \cdot 100.$$

Можно примерно оценить, что оно меньше, чем 100^{100} , поэтому включает не более 200 цифр. Также можно сказать, что на конце этого числа будет 24 нуля, так как нули получаются при умножении на число, кратное 10 (10 штук), при умножении чисел, заканчивающихся на 2 и на 5 (10 штук), при умножении чисел 25, 50, 75 и 100 (содержащих множитель 25) на четные числа (4 дополнительных нуля).

Поскольку $10!$ оканчивается на цифру 8 (если не считать конечные нули), то число $100!$ оканчивается на ту же цифру, что и 8^{10} , то есть на 4 (так как 8^2 заканчивается на 4, 8^5 - на 8).

Проверим правильность выбора основания. Если $d=1000000$, такое значение поместится в ячейку типа long int. Наибольшее число, которое может получиться в результате умножения, равно $1000000 \cdot 100$, что тоже помещается в разрешенный диапазон.

<pre>long int A[40], r, d = 1000000L, s; int i, len = 1, k;</pre>		d основание r остаток len длина числа
<pre>A[0] = 1; for (i = 1; i < 40; i ++) A[i] = 0;</pre>		длинное число = 1
<pre>for (k = 2; k <= 100; k ++) { i = 0; r = 0;</pre>		пока не все разряды обработаны или есть перенос
<pre> while (i < len r > 0) { s = A[i]*k + r;</pre>		умножаем разряд, добавляем перенос
<pre> A[i] = s % d;</pre>		текущий разряд не больше d
<pre> r = s / d;</pre>		перенос в следующий разряд
<pre> i ++; } len = i; }</pre>		
<pre>for (i = len-1; i >= 0; i --) if (i == len-1) printf("%lu", A[i]); else Print6 (A[i]); }</pre>		

Приведенная ниже программа выводит значение **100!** на экран. При выводе очередного элемента массива в процедуре **Print6** все лидирующие нули сохраняются. Для самого старшего элемента лидирующие нули выводить не надо. Один из вариантов процедуры приведен ниже.

```
void Print6 ( long int n )
{
    char s[] = "000000";
    int i = 5;

    while ( n ) {
        s[i] = '0' + n%10;
        n /= 10;
        i --;
    }

    printf("%s", s );
}
```

В этой процедуре мы выделяем с помощью операции «остаток от деления» последовательно все цифры числа, начиная с самой младшей, и записываем в строку **s** соответствующий символ. Поскольку коды цифр в таблице кодов расположены последовательно от '0' до '9', то значение выражения **'0'+k** равно коду цифры **k**.



Многочлены

Как известно, многочленом называют функцию вида

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

Одна из простейших задач — вычисление значения многочлена в заданной точке $x = x_0$. Если выполнять вычисления в соответствии с формулой, каждый раз возводя x в степень (с помощью умножения), она требует

$$\begin{array}{ll} \text{сложений} & n \\ \text{умножений} & 1 + 2 + \dots + (n-1) + n = \frac{n(n+1)}{2} \end{array}$$

Однако можно придумать и более эффективный способ.



Схема Горнера

Представим многочлен в виде

$$f(x) = (\dots(a_n x + a_{n-1})x + a_{n-2})x + \dots a_2)x + a_1)x + a_0$$

При этом для вычисления значения функции при известном **x** требуется

$$\begin{array}{ll} \text{сложений} & n \\ \text{умножений} & n \end{array}$$

что значительно быстрее. Ниже приведена функция на языке Си:

```
void Gorner ( float x, float a[], int n )
{
    float v = 0.;
    for ( int i = n; i >= 1; i -- )
        v = ( v + a[i] ) * x;
    v += a[0];
    return v;
}
```



Последовательности и ряды



Последовательности

Последовательность — это набор элементов, расположенных в определенном порядке.

Будем рассматривать только последовательности чисел. Все числа имеют номера, обычно начиная с 1. Часто последовательность может быть задана формулой n -ого члена. Например, для хорошо известной **арифметической прогрессии** формула n -ого члена имеет вид

$$a_n = a_1 + (n - 1)d$$

где a_1 — начальный элемент, а d — разность. Для **геометрической прогрессии**, соответственно, $a_n = a_1 q^{(n-1)}$, где q — знаменатель прогрессии. В таблице ниже приводятся примеры последовательностей.

№	последовательность	a_n
1	1, 3, 5, 7, ...	$2n-1$
2	1, 3, 7, 15, ...	2^n-1
3	2, 9, 28, 65, ...	n^3+1
4	$1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \dots$	$\frac{1}{n}$
5	$1, \frac{5}{7}, \frac{7}{15}, \dots$	$\frac{2n-1}{2^n-1}$

Наиболее распространенные задачи, связанные с последовательностями:

- определить, сколько членов убывающей последовательности больше некоторого заданного значения
- найти первый член последовательности, удовлетворяющий заданному условию
- найти сумму заданного числа элементов последовательности или сумму элементов, удовлетворяющих заданному условию

Задача. Для последовательности 4 в таблице найти

- 1) сумму элементов, которые больше 10^{-10} ,
- 2) сколько элементов вошло в эту сумму,
- 3) каков первый элемент, меньший 10^{-10} ?

При решении подобных задач особое внимание надо обращать на эффективность вычислений и точность. Если на каждом шаге делать умножение и возведение в степень 2^n , такая программа будет неэффективной.

Можно сделать иначе: ввести две переменные, скажем, u и d , которые будут обозначать $2*n-1$ и 2^n , соответственно. Первая будет с каждым шагом увеличиваться на 2, а вторая — в 2 раза. Ниже приведены две программы, одна из которых работает в 3 раза быстрее другой. В обоих случаях искомые величины находятся в переменных s , n , a .

```

int n;
float a, s;
n = 0; s = 0;

while ( 1 ) {
    n ++;

    a=(2*n-1) / (pow(2,n)-1);

    if ( a < 1.e-10 )break;
    s += a;
}

```

```

int n;
float a, s, u, d;
n = 0; s = 0;
u = 1; d = 2;

while ( 1 ) {
    n ++;

    a = u / (d-1.);

    if(a < 1.e-10)break;
    u += 2; d *= 2;
}

```

📖 Рекуррентные последовательности

В 1202 г. итальянский математик Леонардо Пизанский, известный под именем Фибоначчи, предложил такую задачу:

Задача Фибоначчи. Пара кроликов каждый месяц дает приплод - самца и самку, которые через 2 месяца снова дают такой же приплод. Сколько пар кроликов будет через год, если сейчас мы имеем 1 пару молодых кроликов?

Количество кроликов меняется с каждым месяцем в соответствии с последовательностью

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

которую называют **последовательностью Фибоначчи**. Она задается не общей формулой n -ого члена, а **рекуррентной формулой**, в которой n -ый член выражается через предыдущие. При этом надо определить начальные элементы:

$$a_1 = 1, \quad a_2 = 1,$$

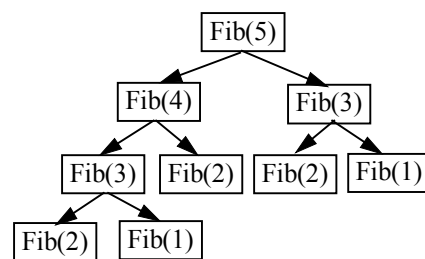
$$a_n = a_{n-1} + a_{n-2} \quad \text{для } n > 2$$

Такая последовательность очень просто реализуется с помощью рекурсивной функции

```

int Fib ( int n )
{
    if ( n < 3 ) return 1;
    else
        return Fib(n-1)+Fib(n-2);
}

```



Однако она крайне неэффективна, потому что вычисление, например, **Fib(5)**, приводит к 9 вызовам функции, что в данном случае неоправданно.

Выход из этой ситуации состоит в написании итеративной функции. Заметим, что для вычисления значения a_n нам надо знать (то есть запомнить) значения a_{n-1} и a_{n-2} (в программе они обозначены как **a1** и **a2**). Чтобы не обрабатывать отдельно случаи, когда $n < 3$, последовательность можно "продолжить влево", добавив фиктивные члены $a_{-1} = 1$ и $a_0 = 0$.

```

int Fib ( int n )
{
    int a2 = 1, a1 = 0, a, i;
    for ( i = 1; i <= n; i ++ )
    {
        a = a1 + a2;
        a2 = a1; a1 = a;
    }
    return a;
}

```

продвижение вперед
на 1 член ряда



Родственные задачи

Задача. Вычислить значение выражения справа.

$$S = \frac{1}{1 + \frac{1}{2 + \frac{1}{\dots + \frac{1}{99 + \frac{1}{100}}}}}$$

Для решения этой задачи заметим, что считать надо снизу. Обозначим

$$s_{100} = \frac{1}{100}, \quad s_{99} = \frac{1}{99 + \frac{1}{100}} = \frac{1}{99 + s_{100}} \quad \text{и т.д.}$$

Числа $s_{100}, s_{99}, s_{98}, \dots, s_1$ представляют собой последовательность с

рекуррентной формулой для n -ого члена $a_n = \frac{1}{101 - n + a_{n-1}}$. Реализация этого алгоритма на

языке Си приводится ниже:

```
float s = 0;
for ( int k = 100; k >= 1; k -- )
    s = 1 / (k + s);
```



Ряды

Часто последовательности, особенно бесконечные, называют **рядами**. Во многих задачах требуется рассчитать **сумму ряда** — бесконечно убывающей последовательности

$$a_1, a_2, \dots \quad |a_n| < |a_{n-1}|$$

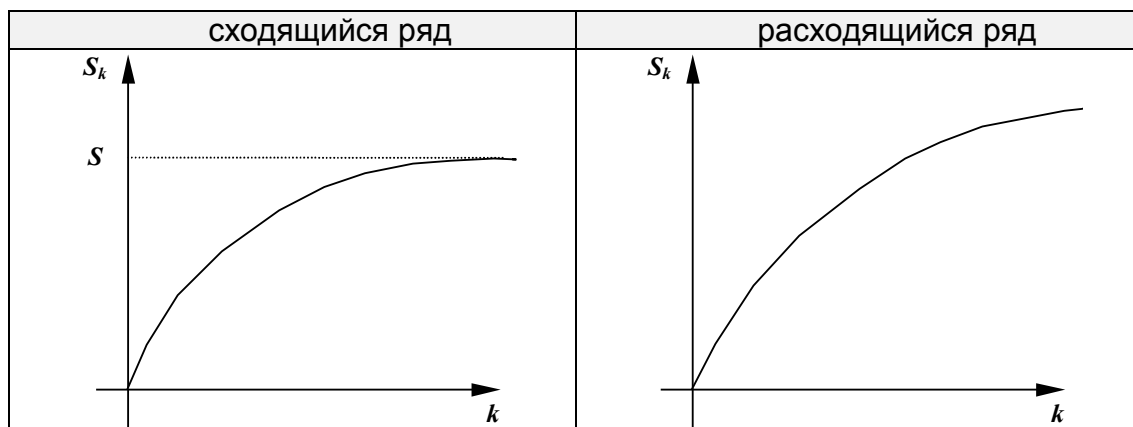
Конечно, на практике невозможно просуммировать бесконечное количество элементов, поэтому бесконечную сумму заменяют конечной — **частичной суммой** ряда, в которую входят первые k элементов:

$$S = \sum_{n=1}^{\infty} a_n \approx S_k = \sum_{n=1}^k a_n$$

Существует два типа рядов: **расходящиеся** и **сходящиеся**. Для расходящегося ряда частичная сумма S_k может стать больше по модулю, чем любое наперед заданное число. Примером расходящегося ряда является **гармонический ряд**:

$$S = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots$$

Для сходящегося ряда частичная сумма стремится, при увеличении количества элементов k , к некоторому фиксированному конечному значению, которое и называется суммой этого ряда.



Для сходящихся рядов имеет смысл и такая задача: найти сумму ряда с заданной точностью ε , так что

$$|S - S_k| = \left| \sum_{k+1}^{\infty} a_n \right| < \varepsilon$$

В общем случае это достаточно сложная математическая задача, однако для рядов особого вида она имеет простое решение.

Если ряд знакопеременный и его элементы убывают, то он сходится и ошибка в вычислении суммы не превышает модуля последнего отброшенного элемента

Знакопеременным называется ряд, в котором знаки элементов чередуются, например

$$S = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \dots + \frac{(-1)^{n+1}}{n} + \dots$$

Вычисление функций

Многие математические функции вычисляются в компьютерах с помощью **функциональных рядов**, то есть рядов, зависящих от переменной. Это позволяет не хранить в памяти таблицы синусов, а непосредственно считать значение функции для любого угла. Как всегда, экономия памяти приводит к снижению быстродействия. Ниже приведены наиболее известные ряды:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots + \frac{x^n}{n!} + \dots$$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots + \frac{(-1)^{n+1} x^{2n-1}}{(2n-1)!} + \dots$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots + \frac{(-1)^{n+1} x^{2n}}{(2n)!} + \dots$$

Для примера напомним функцию для вычисления **sin x** с точностью 10^{-6} . Поскольку ряд для синуса знакопеременный и его элементы убывают, надо суммировать все элементы, модуль которых больше 10^{-6} . Для того, чтобы учитывать меняющийся знак, нет смысла использовать возведение в степень — достаточно ввести переменную, которая будет принимать значение 1 или -1.

```
double sin1 ( double x )
{
    double s = 0, a, xx = x,
           fact = 1, n2 = 1;
    int z = 1;
    do
    {
        a = z * xx / fact;
        s += a;
        z = - z;
        xx *= x*x;
        n2 += 2;
        fact *= (n2-1)*n2;
    }
    while ( fabs(a) > 1.e-6 );
    return s;
}
```

s	частичная сумма ряда
a	элемент ряда
xx	значение x^{2n-1}
fact	значение $(2n-1)!$
n2	значение $2n-1$
z	знак

Интересно определить количество членов ряда, необходимых для вычисления синуса с точностью 10^{-6} , для разных углов (см. таблицу).

диапазон углов (по модулю)	необходимое количество членов ряда
$0^\circ - 90^\circ$	1 — 7
$90^\circ - 180^\circ$	7 — 9
$180^\circ - 270^\circ$	10 — 12
$270^\circ - 360^\circ$	12 — 14

Поскольку синус любого угла может быть представлен как синус угла в интервале от 0° до 90° (возможно с переменной знака)

$$\sin \alpha = \sin(180^\circ - \alpha) = -\sin(-\alpha) = -\sin(\alpha - 180^\circ)$$

в любом случае можно использовать не более 7 членов ряда.



Численное решение уравнений

Многие уравнения невозможно решить аналитически, выписав решение в виде формулы в явном виде. В таких случаях используют **приближенные численные** методы. Приближенными они называются потому, что мы в принципе не можем найти точное решение x^* , однако можем найти некоторое приближение к нему x_0 , которое отличается от точного решения x^* не более, чем на заданную величину ε .

Мы будем рассматривать уравнения вида $f(x)=0$, где $f(x)$ — функция одной переменной. Один из численных методов решения таких уравнений — **метод деления отрезка пополам** или **дихотомии** — мы уже рассматривали. К его преимуществам можно отнести то, что мы можем гарантировать, что ошибка не превышает заданную величину. С другой стороны, надо заранее определить интервал, в котором находится один и только один корень.



Метод хорд

Так же, как и метод дихотомии, этот метод предназначен для уточнения корня на известном интервале $[a, b]$ при условии, что корень существует и функция $f(x)$ имеет на концах отрезка разные знаки.

За следующее приближение к корню принимается не середина отрезка $[a, b]$, как в методе дихотомии, а значение x в точке, где прямая, соединяющая точки $(a, f(a))$ и $(b, f(b))$ пересекает ось ОХ. Уравнение прямой, проходящей через эти точки, имеет вид:

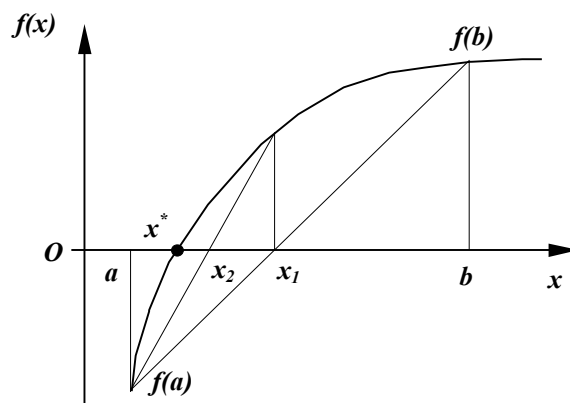
$$\frac{x-a}{b-a} = \frac{y-f(a)}{f(b)-f(a)}$$

В точке пересечения имеем $y=0$. Подставляя в уравнение $y=0$, получаем

$$x_1 = a - \frac{(b-a)f(a)}{f(b)-f(a)}$$

Далее все происходит так, как при использовании метода дихотомии: проверяем, на каком из интервалов $[a, x_1]$ и $[x_1, b]$ находится пересечение, и смещаем соответственно точку a или b .

Более сложен вопрос о том, когда закончить итерации. Обычно применяется два критерия:



1. разность между двумя последовательными приближениями x_k и x_{k-1} стала меньше заданной точности ε , или
2. модуль функции в точке x_k стал меньше заданного значения ε_1 .

При этом, к сожалению, нельзя гарантировать, что ошибка составит не более какой-либо величины — она зависит от вида функции.

```
float Chord( float A, float B, float eps,
            float eps1 )
{
    float x, x0 = A, fA, fB, fX;
    while ( 1 ) {
        fA = F(A); fB = F(B);
        x = A - (B - A) * fA / (fB - fA);
        fX = F(x);
        if ( fabs(fX) < eps1 ) break;
        if ( fA*fX < 0 ) B = x;
        else A = x;
        if ( fabs(x-x0) < eps ) break;
        x0 = x;
    }
    return x;
}
```

следующее
приближение

проверка условия 2

определить интервал для поиска

проверка условия 1

Параметры **eps** и **eps1** обозначают точность для изменения x и значения функции в этой точке. Эта функция может быть еще оптимизирована, так как можно не вычислять заново каждый раз значения $f(a)$ и $f(b)$ — одно из них мы знаем с предыдущего шага.



Метод Ньютона (метод касательных)

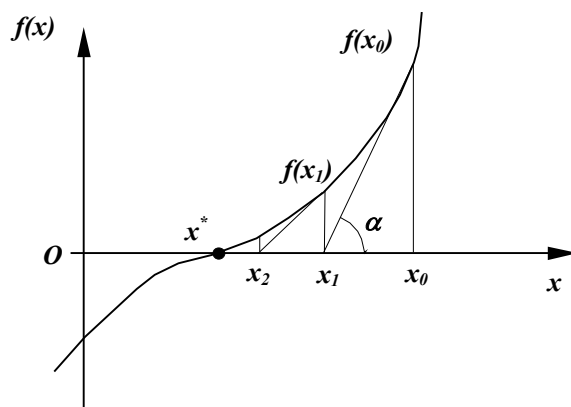
Чаще всего на практике используется метод Ньютона, который обладает быстрой сходимостью и требует знать только начальное приближение x_0 к корню, а не интервал, в котором он находится.

За следующее приближение к корню принимается значение x в точке пересечения касательной, проведенной к кривой в точке $(x_0, f(x_0))$, и оси ОХ. Поскольку

$$\operatorname{tg} \alpha = f'(x_0) = \frac{f(x_0)}{x_0 - x_1},$$

для k -ого шага итерации получаем

$$x_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}$$



Метод Ньютона обладает высокой скоростью сходимости. Обычно точность 10^{-6} достигается за 5-6 итераций. Его главный недостаток в том, что надо на каждом вычислять производную функции, а выражение для нее может быть неизвестно, например, если она задана таблицей. Иногда используют модифицированный метод Ньютона, в котором на всех шагах используется производная, вычисленная в точке x_0 . Важно также правильно выбрать начальное приближение к корню, иначе метод Ньютона может "зациклиться".

Приведенная ниже функция использует функции $F(x)$ и $DF(x)$, которые возвращают значение функции и ее производную в точке x .

```
float Newton ( float x0, float eps)
{
    float f1;
    do {
        f1 = F(x0) / DF(x0);
        x0 -= f1;
    }
    while ( fabs(f1) > eps );
    return x0;
}
```

Метод Ньютона используется также для решения систем нелинейных уравнений с несколькими переменными, но формулы оказываются более сложными и мы их не рассматриваем.



Метод итераций

От исходного уравнения $f(x) = 0$ можно легко перейти к эквивалентному (имеющему те же самые корни)

$$x + bf(x) = x$$

которое получено сначала умножением обеих частей на константу b (не равную нулю), а затем добавлением x . Вводя обозначение $\varphi(x) = x + bf(x)$, получаем уравнение

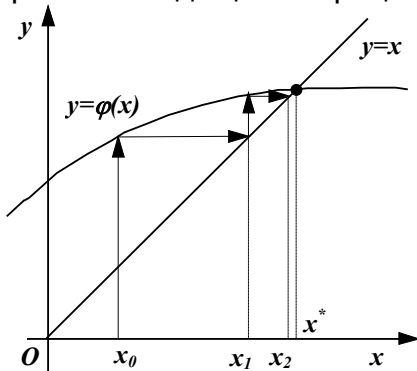
$$x = \varphi(x)$$

которое дает итерационную формулу

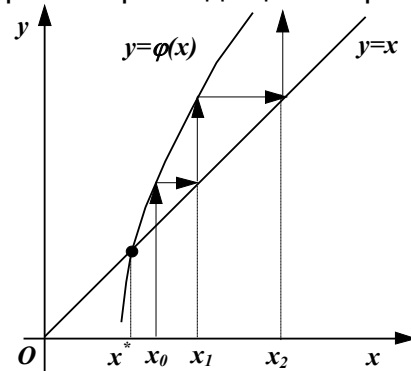
$$x_k = \varphi(x_{k-1}).$$

Теперь надо установить, при каких условиях такая процедура позволяет получить значение x^* (процесс сходится). Рассмотрим 4 случая, изображенные на рисунках.

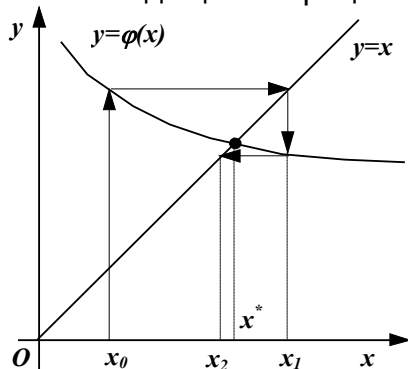
а) односторонний сходящийся процесс



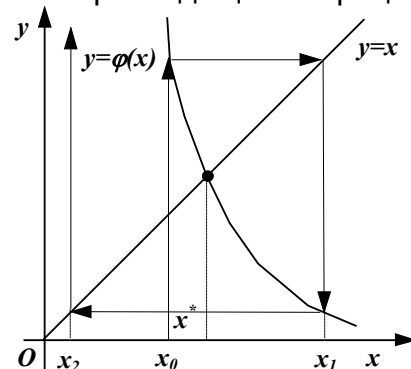
б) односторонний расходящийся процесс



в) двусторонний сходящийся процесс



г) двусторонний расходящийся процесс



Можно показать, что процесс сходится при условии

$$|\varphi'(x)| < 1$$

причем при $0 < \varphi'(x) < 1$ имеет место односторонняя сходимость, а при $-1 < \varphi'(x) < 0$ — двусторонняя. Учитывая, что $\varphi(x) = x + bf(x)$, можно получить

$$\varphi'(x) = 1 + bf'(x)$$

Поскольку наибольшая скорость сходимости наблюдается при $\varphi'(x) = 0$, для этого случая имеем

$$b = -\frac{1}{f'(x)}$$

что дает формулу Ньютона. Таким образом, метод Ньютона имеет наивысшую скорость сходимости среди всех итерационных методов.

Поскольку в общем случае нельзя гарантировать, что метод итераций сходится (это его главный недостаток), в программе надо ограничивать количество шагов некоторым максимальным значением. Функция, реализующая метод итераций, приведена ниже. Выход происходит тогда, когда разность между двумя последовательными приближениями станет меньше заданной точности ε или превышено заданное максимальное число итераций n . Для того, чтобы вызывающая программа могла получить информацию о том, что процесс расходится, параметр n сделан переменным — на выходе он равен числу итераций. Если оно превышает заданное значение, результат неверный и надо менять константу b .

<pre>float Iter (float x0, float eps, int &n) { int i = 0; float x = x0;</pre>		
do {	x0 = x;	запомнить предыдущее x
	x = F(x0);	шаг итерации
	i ++;	
	if (i > n) break;	выход, если процесс расходится
	}	
	while (fabs(x-x0) > eps);	пока решение не найдено
<pre> n = i; return x; }</pre>		

Вызов этой функции может выглядеть так:

<pre>int n = 100; float x; ...</pre>
<pre>x = Iter (1.2, 0.0001, n);</pre>
<pre>if (n > 100) printf("Процесс расходится");</pre>



Использование функции-параметра

В параметры всех процедур можно включать функцию, которая используется в вычислениях. Сначала надо объявить новый тип данных — функция, которая принимает один вещественный параметр и возвращает вещественный результат.

<pre>typedef float (*func)(float x);</pre>
--

Затем этот тип `func` можно использовать в параметрах функций. Например, функция для метода итераций может быть модифицирована так

```
float Iter ( func F, float x0, float eps, int &n)
{
    ...
}
```

Все ее содержимое сохраняется без изменений. При вызове надо передать в качестве первого параметра имя функции, которая вычисляет по заданной формуле.



Вычисление определенных интегралов

Ставится задача вычисления определенного интеграла

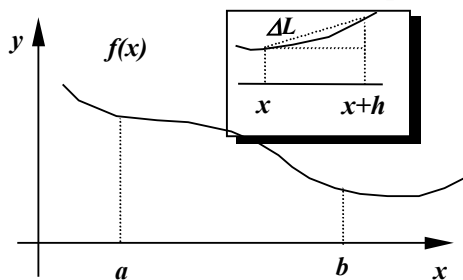
$$J = \int_a^b f(x) dx$$

на конечном интервале в том случае, когда первообразной в аналитическом виде не существует или она очень сложна. Поскольку этот интеграл геометрически представляет собой **площадь** под кривой $y=f(x)$, для его вычисления можно использовать все методы вычисления площадей, рассмотренные в начале главы (методы прямоугольников, трапеций, Монте-Карло, Симпсона).



Вычисление длины кривой

Пусть задана однозначная функция одной переменной $y=f(x)$. Требуется найти длину L кривой, которая задается этой функцией, на известном интервале $[a, b]$.



Эта задача решается в высшей математике с помощью определенного интеграла

$$L = \int_a^b \sqrt{1 + [f'(x)]^2} dx$$

Однако взять такой интеграл аналитически достаточно сложно, поэтому мы будем использовать приближенный метод. Рассмотрим участок кривой на интервале $[x, x+h]$, где h - малая величина шага. Можно приближенно считать, так же, как и при вычислении площади, что длина этой части кривой приближенно равна длине отрезка, соединяющего точки кривой на концах интервала. По теореме Пифагора находим

$$\Delta L = \sqrt{h^2 + [f(x+h) - f(x)]^2}$$

Теперь для определения полной длины кривой на участке $[a, b]$ остается сложить длины всех таких отрезков при изменении x от a до b . Соответствующая функция приведена ниже

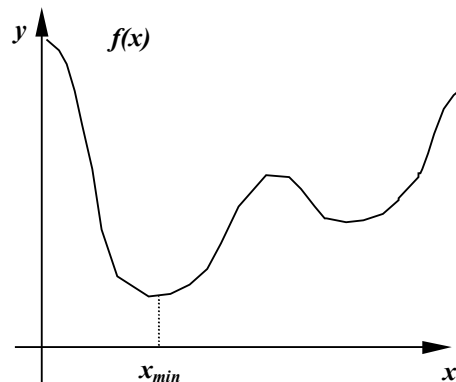
```
float CurveLength ( func F, float a, float b )
{
    float x, dy, h = 0.0001, h2 = h*h, L = 0;
    for ( x = a; x < b; x += h )
    {
        dy = F(x+h) - F(x);
        L += sqrt(h2 + dy*dy);
    }
    return L;
}
```



Оптимизация

На практике часто существует множество решений задачи, и надо найти оптимальное решение, то есть выбрать параметры из некоторой области так, чтобы заданная функция (например, потери) была минимальной. Если надо, найти максимум функции (например, прибыли), надо сменить ее знак на обратный и искать минимум новой функции.

Задача. Найти такое значение параметра x , при котором функция $f(x)$ достигает минимального значения в некоторой области.



Задача осложняется тем, что в реальных задачах существует множество минимумов функции (они называются **локальными**), тогда как нас интересует **глобальный** минимум — минимальное из всех этих значений. Общих способов поиска глобального минимума функций не существует. Для функций одной переменной задача разбивается на 2 этапа:

1. Некоторым способом (например, графически) определяются интервалы, на которых функция $f(x)$ имеет один минимум или максимум, то есть является **униmodalной**.
2. На каждом интервале уточняются минимальные значения функций и из них выбирается наименьший.



Метод золотого сечения

Этот метод предназначен для уточнения минимума функции на интервале $[a, b]$, где она является униmodalной (имеет только один минимум). Задача состоит в том, чтобы использовать такую стратегию поиска, которая позволила бы построить последовательность уменьшающихся интервалов неопределенности

$$[a, b], \quad [a_1, b_1], \quad [a_2, b_2], \quad \dots, \quad [a_n, b_n]$$

так чтобы длина последнего интервала стала меньше допустимой точности ε . Доказано, что оптимальной будет стратегия, при которой используются числа Фибоначчи:

$$F_0 = 1, \quad F_1 = 1, \quad F_k = F_{k-1} + F_{k-2} \quad \text{для } k > 1$$

Интервал $[a, b]$ делится на 2 части в отношении двух последовательных чисел Фибоначчи в точке x_1 . Далее, симметрично относительно центра отрезка выбирается точка x_2 . Следующий интервал определяется в зависимости от значений функции в точках x_1 и x_2 так, как показано в таблице (будем считать, что $x_1 < x_2$).

$f(x_1) < f(x_2) \Rightarrow [a, x_2]$	$f(x_1) > f(x_2) \Rightarrow [x_1, b]$	$f(x_1) = f(x_2) \Rightarrow [x_1, x_2]$

Существенный недостаток этого метода состоит в том, что при разбиении отрезка надо учитывать номер шага N , чтобы найти соответствующие числа Фибоначчи и их отношение F_{N-1}/F_N . Установлено, что при увеличении N это отношение стремится к числу ≈ 0.618034 .

Деление отрезка на две части в таком отношении называется **золотым сечением**. При этом отношение длины большей части отрезка ко всей длине равно отношению длины меньшей части к большей, то есть, обозначив через g длину большей части, получаем

$$g = \frac{1-g}{g},$$

что приводит к квадратному уравнению, положительный корень которого равен

$$g = \frac{-1 + \sqrt{5}}{2} \approx 0.618034$$

Если на каждом шаге отрезок $[a, b]$ делится в этом отношении, после N шагов длина интервала неопределенности уменьшится до $(b-a)g^N$. Это позволяет заранее оценить необходимое количество итерации для достижения заданной точности. Такой метод называется **методом золотого сечения**.

```
typedef float (*func) (float x);

float Gold ( func F, float a, float b, float eps )
{
    float x1, x2, g=0.618034, R = g*(b - a);
    while ( fabs(b-a) > eps ) {
        x1 = b - R;
        x2 = a + R;
        if ( F(x1) > F(x2) ) a = x1;
        else                b = x2;
        R *= g;
    }
    return (a + b) / 2.;
}
```

В приведенном варианте программа ищет минимум функции на заданном интервале. Если надо искать максимум, функцию требуется немного изменить.

Недостаток этого способа в том, что надо знать интервал, в котором находится единственный минимум. Кроме того, его сложно применить для поиска минимума функций нескольких переменных.



Градиентный метод

Для использования этого метода надо знать некоторое начальное приближение к точке минимума. Далее используется следующий принцип: смещаемся на шаг h в ту сторону, в которую функция убывает быстрее всего. Для функции многих переменных это направление определяется **градиентом** — матрицей, составленной из различных производных функции. Для функции одной переменной итерационную формулу можно записать так

$$x_k = x_{k-1} - hf'(x_{k-1})$$

Если производная положительна (функция возрастает), надо уменьшать x , если отрицательна, то увеличивать. При этом шаг h уменьшается до тех пор, пока значение функции в точке x_k не станет меньше, чем $f(x_{k-1})$. Только после этого изменения принимаются и изменяется x . Итерации заканчиваются, когда разность между двумя последовательными значениями x станет меньше по модулю, чем заданная точность ε .

Поскольку необходимо вычислять не только значение функции, но и ее производную, функция Grad, реализующая градиентный метод, принимает в параметрах адреса двух функций, начальное приближение, начальный шаг и требуемую точность.

<pre>float Grad (func F, func DF, float x, float h, float eps) { float fx = F(x), dx; while (1) { dx = - h * DF(x); if (fabs(dx) < eps) break; if (F(x+dx) > fx) h /= 2.; else { x += dx; fx = F(x); } } return x; }</pre>	
новый шаг	← dx = - h * DF(x);
нашли решение	← if (fabs(dx) < eps) break;
шаг неудачный, уменьшаем h	← if (F(x+dx) > fx) h /= 2.;
шаг удачный, запомнили его	← x += dx; fx = F(x);

Приведенная выше функция в принципе не всегда находит минимум (например, если функция бесконечно убывает). Это зависит от выбранного начального приближения x_0 . Кроме того, даже если мы нашли минимум, это не гарантирует, что он глобальный, то есть значение функции в этой точке наименьшее. К ее недостаткам надо отнести и то, что требуется вычислять производную функции, что не всегда легко. Иногда приходится делать это численно, заменяя производную на величину

$$\frac{f(x) - f(x + \Delta x)}{\Delta x}$$

где Δx — достаточно малая величина. Градиентный метод используется главным образом для минимизации функций нескольких переменных.



Оптимизация для функции нескольких переменных

На практике функции, для которых надо искать минимум или максимум, зависят от многих переменных (чем их больше, тем сложнее задача).

Задача. Найти такую комбинацию параметров $\{x, y, \dots\}$, при которой функция $f(x, y, \dots)$ достигает минимального значения в окрестности точки (x_0, y_0) .

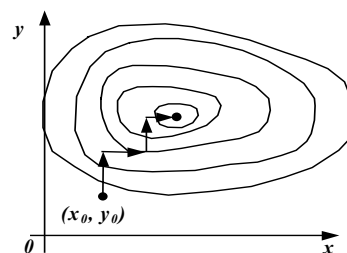
Главная проблема заключается в том, что обычно функция имеет много локальных минимумов и результат зависит от удачного или неудачного выбора начального приближения.

В практических вычислениях используются главным образом следующие три метода оптимизации. Для упрощения мы рассмотрим только случай функции двух переменных без сложных математических выводов.

Метод покоординатного спуска

Выбираем начальное приближение — точку (x_0, y_0) . Сначала фиксируем все координаты, кроме одной. Тогда функция, в которой можно изменять только одну координату, является обычной функцией одной переменной. Ее минимум можно найти, например, с помощью метода золотого сечения.

Изобразим на рисунке *изолинии* — линии равного значения функции двух переменных (как на географических картах). В этом случае минимизация функции при одной изменяемой переменной равносильна движению вдоль одной из осей координат.



Далее сделаем изменяемой другую координату, зафиксировав все остальные. Процедура повторяется до тех пор, пока не будет достигнут минимум, то есть очередной шаг не станет меньше заданной точности.

Градиентный метод

Этот метод связан с математическим понятием *градиента*. **Градиентом** называют вектор, направленный в сторону наискорейшего возрастания функции в данной точке. Следовательно, чтобы найти минимум, надо или в противоположную сторону. Длина шага часто выбирается из условия минимума функции вдоль направления, противоположного градиенту. Такой вариант называется **методом наискорейшего спуска**.

Метод случайного поиска

Метод случайного поиска показал высокую эффективность в сложных задачах, когда применение других методов затруднительно или неэффективно из-за большого количества параметров и сложности вычисления производной функции.

Его идея заключается в следующем: от исходной точки делаем шаг в случайном направлении. Если в новой точке значение функции меньше прежнего, переходим в нее и продолжаем поиск, если больше, то делаем попытку идти в другом (также случайном) направлении. Если сделано значительное число попыток и нигде значение функции не уменьшается в сравнении с исходной точкой, мы находимся вблизи минимума и надо уменьшить длину шага. Процесс заканчивается, когда величина шага становится меньше заданного значения.

4. Моделирование



Что такое модель?

Модель — это упрощенное представление о реальном объекте, процессе или явлении.

Зачем же использовать упрощенные модели и почему не исследовать реальный объект? На это есть несколько причин:

- Реального объекта может уже (или еще) не быть в действительности, например, при моделировании событий прошлого или при разработке нового технического сооружения.
- Проведение экспериментов на реальном объекте очень дорого стоит или может быть опасным и привести к аварии, например, при исследовании подводной лодки или ядерного реактора.
- Иногда надо исследовать только какое-то одно свойство оригинала, а реальный объект имеет много взаимосвязанных свойств.

Моделирование — это построение и исследование моделей объектов и процессов.

Перечислим **цели моделирования**, то есть ответим на вопрос, зачем нужны модели.

- **Познание мира.** Человек стремится понять, как устроен мир, выдвигает и проверяет гипотезы о связях и закономерностях в объектах и процессах.
- **Создание объектов с заданными свойствами.** В этом случае основная цель моделирования — ответить на вопрос, можно ли создать такой объект, и если можно, то как это сделать.
- **Анализ влияния различных воздействий на объект.** Задача моделирования — ответить на вопрос, что будет, если изменить некоторым образом исходные данные или условия.
- **Повышение эффективности управления.** В этом случае модель объекта служит для улучшения характеристик системы, например, для снижения качки судна на волнении.



Виды моделей

Модели можно классифицировать по различным признакам. Один из наиболее важных — способ представления модели. Различают

- **материальные (физические)** модели, которые "можно потрогать", они отражают геометрические и физические свойства оригинала в увеличенном, уменьшенном или просто упрощенном виде; к таким моделям относятся, например, детские игрушки;
- **вербальные (словесные)** модели — информационные модели в мысленной или разговорной форме;
- **структурные** модели — схемы, графики, таблицы;
- **логические** модели, которые описывают выбор одного из вариантов на основе анализа условий;
- **математические** модели — формулы, отображающие связь параметров объектов и процессов.

Нас будет интересовать математическое моделирование. Математические модели бывают **статические** (не учитывающие изменение характеристик объекта во времени) и **динамические**. Наиболее сложны **динамические модели**, которые во многих случаях описываются

нелинейными дифференциальными уравнениями, для которых не существует аналитического решения и надо применять численные методы и моделирование. Моделирование обязательно используется при разработке любых достаточно сложных устройств и в научных исследованиях в двух случаях:

1. Если нельзя получить аналитическое решение задачи (формулу).
2. Эксперимент поставить невозможно, или дорого, или опасно.

Иногда применяют моделирование на уменьшенных моделях, иногда — численное математическое моделирование с помощью вычислительной машины.



Вращение

Один из самых сложных элементов анимации — вращение фигуры относительно некоторого центра. Мы рассмотрим самый простой случай вращения, когда объект — окружность.



Предварительный анализ

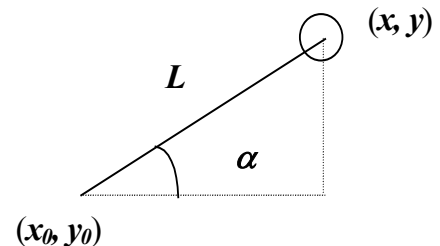
Пусть цент вращеия задается координатами x_0 и y_0 , а координаты центра шарика — x и y и радиус орбиты (окружности, по которой вращается шарик) равен L . Тогда при известных x_0 , y_0 , L и угле α , координаты x и y могут быть вычислены из прямоугольного треугольника

$$x = x_0 + L \cos(\alpha)$$

$$y = y_0 - L \sin(\alpha)$$

Независимой переменной в данном случае удобно выбрать угол α , а не координаты объекта. За 1 шаг он изменяется на $\Delta\alpha$. Положительная величина $\Delta\alpha$ задает вращение против часовой стрелки, отрицательная — по часовой.

Такой подход приводит к тому, что положение фигуры фактически задается в полярных координатах с полюсом в центре вращения. Для вращающихся объектов это наиболее естественный и простой способ задания координат.



Программа

<pre>#include <graphics.h> #include <conio.h> #include <dos.h> #include <math.h></pre>	процедура рисует цветом c фигуру с координатами (x,y)
<pre>void Figure (int x, int y, int color);</pre>	
<pre>void main() { int drv = VGA, mode = VGAHI; int x0, y0, x, y, L; float a, da; initgraph (&drv, &mode, "c:\\borlandc\\bgi"); x0 = 320; y0 = 240; L = 100; a = 0; da = 1. * M_PI / 180.;</pre>	изменение угла за 1 шаг цикла
<pre>while (1) { if (kbhit()) if (getch() == 27) break;</pre>	выход по Esc
<pre> x = x0 + L * cos(a); y = y0 - L * sin(a);</pre>	новые координаты
<pre> Figure (x, y, YELLOW); delay (10); Figure (x, y, BLACK); a += da;</pre>	изменение угла поворота
<pre> } closegraph(); }</pre>	

В программе переменная **a** обозначает угол α , а переменная **da** – изменение угла за 1 шаг цикла. Так как **da** не меняется, то вращение происходит с постоянной угловой скоростью.

Вращение с остановкой

Сделаем вращение замедленным так, чтобы скорость упала до нуля ровно за 5 секунд. Будем считать, что величина задержки при вызове процедуры `delay` равна 10 мс. Тогда за 5 секунд должно выполняться примерно $5000/10=500$ шагов цикла. Мы введем еще одну переменную **dda**, в которой записано изменение скорости вращения, то есть изменение **da**, за 1 шаг цикла. Чтобы скорость упала до нуля ровно за 5 секунд, надо на каждом шаге вычитать из **da** величину, в 500 раз меньшую, чем начальная скорость вращения. Чтобы не началось вращение в обратную сторону, при получении отрицательной скорости будем устанавливать ее в ноль. Основные изменения в программе выглядят так:

<pre>float a, da, dda; . . . a = 0; da = 1. * M_PI / 180.;</pre>	
<pre>dda = - da / 500;</pre>	уменьшение скорости вращения за 1 шаг
<pre>while (1) { . . . a += da;</pre>	
<pre>da += dda;</pre>	изменение скорости вращения
<pre>if (da < 0) da = 0;</pre>	остановка
<pre>}</pre>	



Использование массивов

Во многих задачах моделирования и при создании игр надо использовать массивы. В этом примере мы смоделируем кипение воды. Пузырьки воды начинают подниматься со дна и лопаются, когда доходят до поверхности.



Предварительный анализ

Мы будем считать, что в любой момент в кастрюле (размером с экран) находится 100 пузырьков и когда какой-то из них лопается (доходит до верхней кромки экрана), вместо него появляется новый пузырек на дне.

Для того, чтобы хранить координаты пузырьков (x и y), надо использовать два массива по 100 элементов каждый. Перенесем все основные операции в процедуры так, чтобы в основной программе оставить только логику (последовательность действий).

В начале программы надо записать в массивы X и Y координаты пузырьков так, чтобы они равномерно заполняли весь экран (мы будем считать, что ведем наблюдение через некоторое время после начала кипения). Это можно сделать например, с помощью случайных чисел.

Основной цикл должен выглядеть так:

- Проверяем, не нажата ли клавиша Esc, если да, то выходим из цикла.
- Рисуем все пузырьки на экране.
- Делаем задержку, например, 10 мс.
- Стираем пузырьки (то есть, рисуем их черным цветом).
- Сдвигаем все пузырьки вверх на заданное число пикселей.
- Проверяем, не вышли ли пузырьки за верхнюю границу экрана; если да, то создаем новые пузырьки на дне вместо лопнувших.

Мы будем использовать три процедуры:

Draw - рисование всех пузырьков заданным цветом

Sdvig - сдвиг всех пузырьков вверх

Zamena - проверяем выход за границу экрана и создаем новые пузырьки вместо лопнувших.



Основная программа

```
#include <graphics.h>
#include <conio.h>
#include <dos.h>
#include <stdlib.h>
```

Глобальные переменные, они
доступны всем процедурам

```
#define N 100
```

```
int X[N], Y[N], r = 3;
```

```
void Draw ( int color );
void Sdvig ( int dy );
void Zamena ( );
```

Объявления
вспомогательных процедур

```
void main()
{
    int drv = VGA, mode = VGAHI;
    initgraph ( &drv, &mode, "c:\\borlandc\\bgi" );
```

```
    int i;
    for ( i = 0; i < N; i ++ )
    {
        X[i] = random(640 - 2*r) + r;
        Y[i] = random(480 - 2*r) + r;
    }
```

Случайные координаты
первых пузырьков

```
    while ( 1 )
    {
        if ( kbhit() )
            if ( getch() == 27 ) break;
```

```
        Draw ( YELLOW );
        delay ( 10 );
        Draw ( BLACK );
        Sdvig ( 4 );
        Zamena();
```

```
    }
    closegraph();
```

```
}
```

Массивы **X** и **Y** необходимо использовать во всех процедурах и в основной программе. Существует два способа сделать это: передавать их в процедуры в качестве параметров или сделать **глобальными**, то есть доступными всем процедурам.

Глобальными называются переменные и массивы, доступные всем процедурам. Глобальные переменные объявляются вне всех процедур и функций, обычно в самом начале файла. При создании они заполняются **нулями**.

В список глобальных переменных мы также включим и радиус пузырьков **r**.

До начала основного цикла мы заполняем массивы координат случайными значениями. Нам надо, чтобы все пузырьки находились в границах экрана. Поэтому, например, координаты центра пузырька **x** и **y** должны быть в границах

$$r \leq x \leq 640 - r, \quad r \leq y \leq 480 - r,$$

Именно такие диапазоны обеспечиваются формулами

```
X[i] = random(640 - 2*r) + r;
Y[i] = random(480 - 2*r) + r;
```

Основной цикл содержит вызовы процедур. За 1 шаг цикла пузырьки сдвигаются вверх на 4 пиксела, это число указано в скобках при вызове процедуры Sdvig.

Вспомогательные процедуры

```
void Draw ( int color )
{   int i;
    setcolor ( color );
    for ( i = 0; i < N; i ++ )
        circle ( X[i], Y[i], r );
}

void Sdvig ( int dy )
{   int i;
    for ( i = 0; i < N; i ++ ) Y[i] -= dy;
}

void Zamena ()
{   int i;
    for ( i = 0; i < N; i ++ )
        if ( Y[i] <= r ) {
            X[i] = random(640 - 2*r) + r;
            Y[i] = 480 - r;
        }
}
```

В процедуре Draw мы рисуем все пузырьки. Их радиус одинаковый и равен r , координаты центров находятся в массивах X и Y , а цвет задается как параметр процедуры. Черный цвет означает стирание. Процедура Sdvig уменьшает все координаты y на величину, которая задана в виде параметра. В процедуре Zamena проверяем в цикле координату y для каждого пузырька и, если она меньше r , создается новый пузырек на дне кастрюли.

Математическое моделирование физических процессов

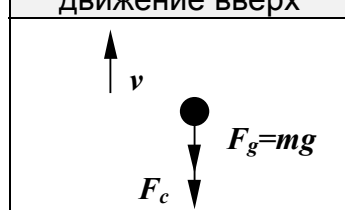
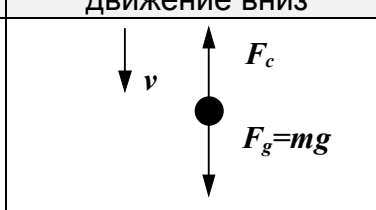
Постановка задачи

Вертикально вверх со скоростью $v_0 = 60$ м/с вылетает свинцовый шарик (плотность свинца $\rho = 11350$ кг/м³) диаметром 10 мм. Определить

1. через какое время и с какой скоростью он вернется обратно;
2. на какую максимальную высоту он поднимется.

Построение полной математической модели

На этом этапе наша задача — построить как можно более полную математическую модель задачи. Основные силы при движении шарика вверх и вниз показаны на рисунках.

движение вверх	движение вниз
	

Здесь F_g — сила тяжести, которая рассчитывается через массу шарика $m = \rho V$, где ρ — плотность свинца, а V — объем шарика, равный

$$V = \frac{4}{3} \pi R^3$$

где R — радиус шарика. Через F_c обозначена сила сопротивления воздуха. Если скорость шарика не превышает 100 м/с, она может быть рассчитана по формуле Стокса

$$F_c = -6\pi\eta Rv,$$

где v — скорость шарика, а η — динамическая вязкость воздуха (Па·с). Вязкость воздуха зависит от высоты, температуры, влажности, ветра и многих других параметров. Обычно в расчетах принимается $\eta \approx 0,022$ Па·с. Знак минус в формуле указывает на то, что направление действия силы сопротивления всегда противоположно направлению вектора скорости.

Кроме того, на шарик действует ветер, но не очень ясно, как его учитывать.

Введение допущений

Полные математические модели очень редко используются при моделировании. Причина в том, что некоторые незначительные влияния (например, ветер) просто невозможно учесть и ими пренебрегают. Кроме того, учет всех факторов может недопустимо увеличить объем вычислений. В данном примере вводим следующие допущения:

1. Ускорение свободного падения постоянно.
2. Масса и диаметр шарика не меняются.
3. Влияние ветра не учитывается.
4. Формула Стокса справедлива, причем вязкость — величина постоянная и равна $\eta \approx 0,022$ Па·с.

Проверим, нельзя ли вообще пренебречь сопротивлением воздуха. Если его нет, то имеем из баланса энергии:

$$\frac{mv_0^2}{2} = mgh_{\max}^0, \quad h_{\max}^0 = \frac{v_0^2}{2g} = \frac{60^2}{2 \cdot 9.81} = 183,5 \text{ м}$$

Общее время полета равно удвоенному времени падения с высоты h_{\max}^0 :

$$t_{\max}^0 = 2\sqrt{\frac{2h_{\max}^0}{g}} = 12,2 \text{ с}$$

Максимальная скорость равна 60 м/с.

Теперь оценим, что изменится при учете сопротивления воздуха. Явно, что максимальная высота уменьшится. При обратном падении скорость стабилизируется, когда сила сопротивления станет равна силе тяжести, то есть

$$6\pi\eta Rv = mg = \rho \frac{4}{3} \pi R^3 g$$

Отсюда следует, что $v_{\max} < \frac{\rho \frac{4}{3} \pi R^3 g}{6\pi\eta R} = 28,11$ м/с. Это означает, что при вычислении скорости мы ошиблись по крайней мере в 2 раза. Поэтому сопротивлением воздуха пренебрегать нельзя.

Дискретизация непрерывных процессов

Начальные условия. $h = 0, \quad v = v_0 = 60 \text{ м/с.}$

Характер движения. Про движение шарика с учетом сопротивления воздуха нельзя сказать, что оно равномерное или хотя бы равнозамедленное, поскольку скорость меняется, а сила сопротивления воздуха зависит от скорости, таким образом, ускорение не постоянно.

Хотя в каждый момент действующие силы, скорость и ускорение шарика меняются, для моделирования мы считаем, что на каждом очень маленьком интервале времени Δt действующие силы и, следовательно, ускорение, постоянно.

При дискретизации все время движения разбивается на малые интервалы длительностью Δt , такие, что внутри них все силы можно считать примерно постоянными.

Пусть в момент $t_i = i \cdot \Delta t$ известны скорость v_i и высота h_i . Положительной будем считать скорость шарика при движении вверх. Тогда ускорение определится разностью сил при данной скорости:

$$a_i = \frac{-6\pi\eta R v_i - mg}{m} = \frac{-6\pi\eta R v_i}{m} - g$$

Тогда, считая, что $a = a_i = \text{const}$ на интервале $[t_i, t_{i+1}]$, по формулам равноускоренного движения рассчитываем скорость и ускорение в конце этого интервала:

$$v_{i+1} = v_i + a_i \cdot \Delta t$$

$$h_{i+1} = h_i + v_i \cdot \Delta t + \frac{a_i \cdot \Delta t^2}{2}$$

Таким образом, последовательно рассчитываем все параметры движения шарика до тех пор, пока не получим нулевую (или отрицательную) высоту — это означает, что шарик вернулся обратно.

Остается решить единственную проблему — выбрать **интервал дискретизации Δt** . Общих методов решения этой задачи нет. Очевидно, что увеличение интервала квантования ускоряет вычисления и снижает ошибку моделирования и наоборот. Обычно, если из физических соображений неясно, каков должен быть интервал квантования, проводится серия вычислительных экспериментов, в ходе которой его значение постепенно уменьшается до тех пор, пока результаты моделирования не стабилизируются.



Составление программы

Ниже приводится программа, с помощью которой моделируется движение свинцового шарика, брошенного вертикально вверх. В результате она выведет отрицательную скорость в соответствии с принятым направлением.

```
float ro = 11350., R=5.e-3, eta = 0.022,
      g = 9.81;
float m = ro*4./3.*M_PI*R*R*R, v0 = 60;
float Fc, dt = 0.001, a, v, t, h, hmax = 0;

v = v0; t = 0; h = 0.;
while ( h >= 0 ) {
    Fc = - 6*M_PI*eta*R*v;
    a = Fc/m - g;
    h += v*dt + a*dt*dt;
    v += a*dt;
    t += dt;
    if ( h > hmax ) hmax = h;
}
printf("\nH=%f v=%f t=%f", hmax, v, t );
```

Для этих данных получаем

$$h_{\max} = 79.87 \text{ м}, \quad v = 23.61 \text{ м/с}, \quad t = 8.52 \text{ с}.$$

5. Сложные проекты

Зачем нужны проекты?

Пока вся наша программа была записана в одном файле и не использовала никаких нестандартных библиотек. Реальные программы включают несколько тысяч строк кода и используют дополнительные функции, записанные в библиотеках, в них используются *проекты*. Таким образом, проекты служат для того, чтобы

- разбить программу на несколько модулей, каждый из которых записывается в отдельный файл (при изменении одного модуля компилируется заново только он). Рекомендуется делать так, чтобы длина каждого модуля была не более 200-300 строк (иначе становится сложно искать в нем нужную функцию или процедуру).
- подключить функции, помещенные в нестандартные библиотеки

Как создать проект?

Сначала создадим простейший проект в который будет входить всего один файл. Для этого надо выполнить следующие действия:

- нажать клавишу **F10**, чтобы перейти в верхнее меню
- выбрать **Project - Open project** (Проект — Открыть проект)
- ввести имя проекта и нажать клавишу **Enter**

После этого вы увидите окно проекта в нижней части экрана. В каждой строчке этого окна записывается имя файла, включенного в проект. В окне проекта вы можете

Добавить в проект новый файл	Нажать клавишу Insert , с помощью диалога выбора файла найти файл на диске и выбрать его
Удалить файл из проекта	Выделить имя файла курсором и нажать клавишу Delete
Открыть файл для редактирования	Выделить имя файла курсором и нажать клавишу Enter
Изменить опции для файла	Нажать клавиши Ctrl-O , установить нужные режимы для файла и нажать клавишу Enter
Вывести список заголовочных файлов	Выделить имя файла курсором и нажать пробел

В проекте сохраняются все настройки, которые вы будете в дальнейшем менять. Это означает, что они будут работать только для одного вашего проекта и не влияют на остальные проекты.

В проект для работы под управлением операционной системы **MS-DOS** можно включать файлы следующих типов

- *.c — исходные тексты программ на языке Си
- *.cpp — исходные тексты программ на языке Си++
- *.obj — объектные файлы
- *.lib — нестандартные библиотеки функций и процедур
- *.asm — исходные тексты программ на языке Ассемблер

Для запуска проекта на выполнение надо нажать на клавиши **Ctrl-F9**. При этом не играет роли, в каком окне вы сейчас работаете (если вы не используете проект, для запуска программы надо обязательно перейти в то окно, в котором находится ее исходный текст).

При выходе из оболочки **Borland C** файл проекта сохраняется на диске в виде файла с расширением *.prj.



Как загрузить проект?

Существует несколько способов загрузки проекта. Пусть есть проект `primer.prj` и нам надо загрузить его в оболочку **Borland C**. Для этого можно выбрать один из следующих вариантов:

1. Набрать в командной строке

```
bc primer.prj
```

2. Запустить оболочку из командной строки, войти в верхнее меню по клавише **F10**, выбрать **Project-Open project**, выделить нужный файл проекта в окне выбора файла и нажать **Enter**.
3. В оболочке **Volkov Commander** выделить курсором нужный файл проекта и нажать на клавишу **Enter** (при этом надо, чтобы оболочка была настроена нужным образом).

Чтобы во время редактирования файла перейти в окно проекта, надо выбрать в верхнем меню **Window-Project** (или нажать клавиши **Alt-W** и затем **P**).



Общие глобальные переменные

Мы уже говорили о том, что одна из главных задач проекта — разбить большую программу на несколько частей, каждую из которых можно отлаживать отдельно (этим могут заниматься даже разные люди). При этом возникают некоторые сложности, если функции и процедуры в разных **модулях** (так называют отдельные файлы, входящие в проект) должны использовать общие глобальные данные или структуры данных. Чтобы в таких ситуациях избежать ошибок, надо помнить два простых правила:

1. В одном модуле (там, где выделяется память под глобальные переменные) они объявляются также, как и обычно.
2. Во всех остальных модулях, использующих глобальные переменные, перед их объявление ставится ключевое слово `extern`. Это означает, что они располагаются в другом модуле.

Например, проект содержит основную программу в модуле `main.cpp` и библиотеку функций в файле `func.cpp`, которые должны использовать глобальную переменную **mode** и глобальный массив целых чисел **X[20]**. Тогда начало текста этих модулей может быть такое:

```
//*****
// Основной модуль main.cpp
//*****
int mode = 0;
int X[20];
void main()
{
    ...
}
```

```
//*****
// Модуль функций func.cpp
//*****
extern int mode;
extern int X[20];
...
```

Помните, что начальные значения глобальных переменных можно задавать только в том модуле, где они размещаются в памяти (то есть объявляются без ключевого слова `extern`).



Общие заголовочные файлы

Пусть наш проект состоит из основного модуля и нескольких модулей с процедурами и функциями, в которых используются глобальные переменные. Как вы знаете, любые переменные до использования необходимо объявить. Так же любые процедуры и функции надо объявить в каждом модуле до того, как их вызовы встречаются в тексте программы. Конечно, можно в начале каждого модуля вписать объявления всех глобальных переменных с ключевым словом `extern` и объявления всех функций, которые используются в этом модуле. Однако чаще всего поступают иначе.

Практически любая программа начинается с директивы `#include`, с помощью которой к модулю подключается заголовочный файл с расширением `*.h`. Если найти этот файл на диске (заголовочные файлы находятся в каталоге **C:\BORLANDC\INCLUDE**) и посмотреть его любым текстовым редактором, вы увидите, что он содержит как раз объявления глобальных переменных и функций.

Язык Си позволяет вам создавать свои заголовочные файлы и подключать их к проекту. В заголовочный файл обычно включают объявления всех глобальных переменных (с ключевым словом `extern`) и объявления всех функций, используемых в проекте. Можно, конечно, записать ваш собственный заголовочный файл в каталог **C:\BORLANDC\INCLUDE** и подключать его так же, как и стандартные файлы. Тем не менее, лучше не путать "свою шерсть с государственной" и свои файлы с системными. Заголовочные файлы записывают в тот же каталог, в котором находятся все остальные файлы проекта, но при подключении их имена записываются не в угловых скобках, а в кавычках, что означает "искать в текущем каталоге".



Пример проекта

Рассмотрим проект, в котором один главный модуль `main.cpp` и два модуля с функциями `simple.cpp` и `draw.cpp`. Все они используют глобальный массив целых чисел **`x[20]`** и глобальную целую переменную **`mode`**. С учетом сказанного выше, заголовочный файл можно составить так:

```
//*****
//      MAIN.H — файл заголовков
//*****

//  объявление глобальных переменных
extern int mode;
extern int X[20];

//  объявление функций
void Draw ( int x, int y);
int Simple ( int N );
```

Для того, чтобы все модули проекта могли использовать эти переменные и функции, в начало каждого из них надо вставить строчку

```
#include "main.h"
```

Надо помнить, что в каком-то одном (и только одном) модуле все глобальные переменные должны быть объявлены без ключевого слова `extern`. Также каждая объявленная функция должна быть определена в одном (и только одном) из модулей.



Оверлейные программы

В очень больших проектах иногда возникает такая ситуация: для работы программы не хватает памяти. В то же время какие-то модули нужны не всегда, а только на некоторое короткое время, например для вывода на экран заставки при старте программы. В этом случае можно попытаться освободить память, построив **оверлейную программу**.

Оверлейная программа - это такая программа, в которой некоторые модули загружаются в память только тогда, когда они используются, и сразу после завершения работы освобождают память.

Чтобы сделать хорошую оверлейную программу, надо

- все функции, которые нужны только в определенный момент, сгруппировать в один модуль
- войти в окно проекта, выделить название этого модуля и нажать **Ctrl-O**
- в окне установки режимов включить переключатель **Overlay this module**

При создании программ для операционной системы **Windows** аналогами оверлейных модулей являются динамически загружаемые библиотеки (DLL).

Глава IV.

Динамические структуры данных

Глава IV. Динамические структуры данных	1
1. Списки	2
Динамические структуры данных	2
Связанный список	2
Операции со списком	3
Двусвязный список	6
Операции с двусвязным списком	7
Циклические списки	9
2. Стеки, очереди, деки	10
Стек	10
Реализация стека в языке Си	10
Системный стек в программах	12
Очередь	12
Дек	12
3. Деревья	14
Что такое деревья?	14
Реализация деревьев в языке Си	16
Сортировка и поиск с помощью дерева	18
Разбор арифметического выражения	20
Дерево игр	25
4. Графы	27
Основные понятия	27
Задача Прима-Краскала	28
Кратчайший путь	30
Оптимальное размещение	32
Задача коммивояжера	34
Задача о паросочетаниях	38

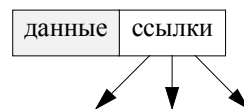
1. Списки



Динамические структуры данных

Часто в программах надо использовать такие типы данных, размер и структура которых должны меняться в процессе работы. Динамические массивы здесь не выручают, поскольку заранее нельзя сказать, сколько памяти надо выделить - это выясняется только в процессе работы. Например, надо проанализировать текст и определить, сколько каких слов в нем встретилось, а также расставить их по алфавиту.

В таких случаях применяют данные особой структуры, которые представляют собой отдельные элементы, связанные с помощью **ссылок**. Каждый элемент (**узел**) состоит из двух областей памяти: **поля данных** и **ссылок**:

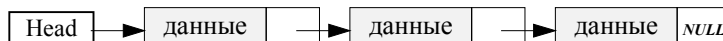


Ссылки представляют собой адреса других узлов этого же типа, с которыми данный элемент логически связан. В языке Си для организации ссылок используются переменные-указатели. При добавлении нового узла в такую структуру выделяется новый блок памяти и устанавливаются связи этого элемента с уже существующими (с помощью ссылок). Для обозначения конечного элемента в цепи используются **нулевые ссылки** (**NULL** в языке Си).



Связанный список

В простейшем случае каждый узел содержит всего одну ссылку. Для определенности будем считать, что решается задача частотного анализа текста - определения всех слов, встречающихся в тексте и их количества. В этом случае область данных элемента включает строку (длиной не более 40 символов) и целое число.



Каждый элемент содержит также ссылку на **следующий** за ним элемент. У последнего в списке элемента поле ссылки содержит **NULL**. Чтобы не потерять список, мы должны хранить адрес его первого узла - он называется "головой" списка. В программе надо объявить два новых типа данных - узел списка **Node** и указатель на него **PNode**. Узел представляет собой структуру, которая содержит три поля - строку, целое число и указатель на такой же узел. Правилами языка Си допускается объявление

```
struct Node {
    char word[40];
    int count;
    Node *next;
};

typedef Node *PNode;
```

В дальнейшем мы будем считать, что указатель **Head** указывает на начало списка, то есть объявлен в виде

```
PNode Head;
```




Операции со списком

Основными операциями являются добавление элемента в список и исключение из списка. Заметьте, что возможно два типа добавления - до и после заданного элемента.



Создание узла

Для того, чтобы добавить узел к списку, необходимо создать его, то есть выделить память под узел и запомнить адрес выделенного блока. Будем считать, что надо добавить к списку узел, соответствующий новому слову, которое записано в переменной **NewWord**. Составим функцию, которая создает новый узел в памяти и возвращает его адрес. Обратите внимание, что при записи данных в узел используется обращение к полям структуры через указатель.

```
PNode CreateNode ( char NewWord[] )
{
    PNode NewNode = new Node;
    strcpy(NewNode->word, NewWord);
    NewNode->count = 1;
    NewNode->next = NULL;
    return NewNode;
}
```

указатель на новый узел

записать данные

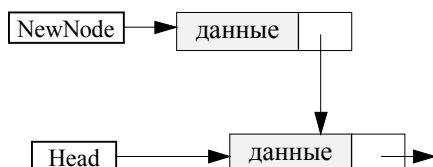
После этого узел надо добавить к списку.



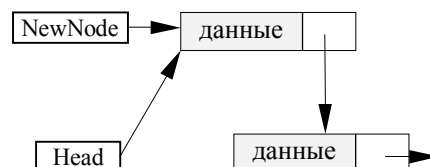
Добавление узла в начало списка

При добавлении нового узла **NewNode** в начало списка надо 1) установить ссылку узла **NewNode** на голову существующего списка и 2) установить голову списка на новый узел.

1)



2)



По такой схеме работает процедура **AddFirst**. Предполагается, что адрес начала списка хранится в глобальной переменной **Head**.

```
void AddFirst(PNode NewNode)
{
    NewNode->next = Head;
    Head = NewNode;
}
```



Проход по списку

Другие операции с односвязным списком требуют поиска заданного элемента в списке, то есть надо просмотреть все элементы пока не найдется нужный нам. Надо учесть, что требуемого элемента может и не быть, тогда просмотр заканчивается при достижении конца списка. Такой подход приводит к следующему алгоритму:

1) начать с головы списка;

- 2) пока текущий элемент существует (указатель - не **NULL**), проверить нужное условие и перейти к следующему элементу;
- 3) закончить работу когда найден требуемый элемент или все элементы списка просмотрены.

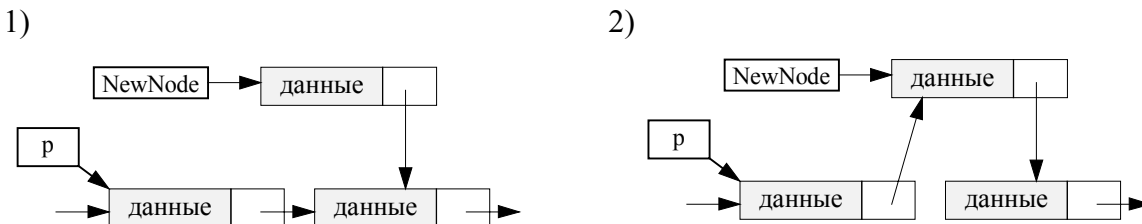
Например, следующая процедура ищет в списке элемент, соответствующий заданному слову (для которого поле **word** совпадает с заданной строкой **NewWord**), и возвращает его адрес или **NULL**, если такого узла нет.

```
void Find (PNode Head, char NewWord[])
{
    PNode q = Head;
    while (q && strcmp(q->word, NewWord))
        q = q->next;
    return q;
}
```

Добавление узла после заданного

Дан адрес **NewNode** нового узла и адрес **p** одного из существующих узлов в списке. Требуется вставить в список новый узел после узла с адресом **p**. Эта операция выполняется в два этапа:

- 1) установить ссылку нового узла на узел, следующий за данным;
- 2) установить ссылку данного узла **p** на **NewNode**.



Последовательность операций менять нельзя, потому что если сначала поменять ссылку у узла **p**, будет потерян адрес следующего узла.

```
void AddAfter (PNode p, PNode NewNode)
{
    NewNode->next = p->next;
    p->next = NewNode;
}
```

Добавление узла перед заданным

Эта схема добавления самая сложная. Проблема заключается в том, что в односвязном списке мы можем, зная адрес узла, получить адрес предыдущего узла, только пройдя весь список сначала. Задача сведется либо к вставке узла в начало списка (если заданный узел - первый), либо к вставке после заданного узла.

```

void AddBefore(PNode p, PNode NewNode)
{
    PNode q = Head;

    if (Head == p) {
        AddFirst(Head, NewNode);
        return;
    }

```

ищем узел, следующий за
которым - **p**

```

    while (q && q->next != p) q = q->next;

```

```

    if (q) AddAfter(q, NewNode);
}

```

выполнять только если нашли узел **p**

Такая процедура обеспечивает "защиту от дурака" - если задан узел, не присутствующий в списке, ничего не происходит.

Существует еще один интересный прием: если надо вставить новый узел **NewNode** до заданного узла **p**, вставляют узел **после** этого узла, а потом выполняется обмен данными между узлами **NewNode** и **p**. Таким образом, по адресу **p** в самом деле будет расположен узел с новыми данными, а по адресу **NewNode** - с теми данными, которые были в узле **p**, то есть мы решили задачу. Этот прием не сработает, если адрес нового узла **NewNode** запоминается где-то в программе и потом используется, поскольку по этому адресу будут находиться другие данные.

Добавление узла в конец списка

Для решения задачи надо сначала найти последний узел, у которого ссылка равна **NULL**, а затем воспользоваться процедурой вставки после заданного узла. Отдельно надо обработать случай, когда список пуст.

```

void AddLast(PNode NewNode)
{
    PNode q = Head;

    if (Head == NULL) {
        AddFirst(Head, NewNode);
        return;
    }

```

список пуст

ищем последний элемент

```

    while (q->next) q = q->next;

```

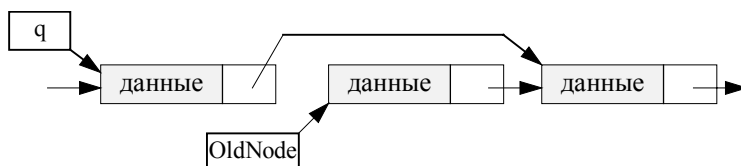
```

    AddAfter(q, NewNode);
}

```

Удаление узла

Эта процедура также связана с поиском заданного узла по всему списку, так как нам надо поменять ссылку у предыдущего узла, а перейти к нему непосредственно невозможно. Если мы нашли узел, за которым идет удаляемый узел, надо просто переставить ссылку.



Отдельно обрабатывается случай, когда удаляется первый элемент списка. При удалении узла освобождается память, которую он занимал.

```

void DeleteNode(PNode OldNode)
{
    PNode q = Head;
    if (Head == OldNode)
        Head = OldNode->next;
    else {
        while (q && q->next != OldNode)
            q = q->next;
        if (q == NULL) return;
        q->next = OldNode->next;
    }
    delete OldNode;
}

```

удаляется первый элемент

ищем
предыдущий
элемент

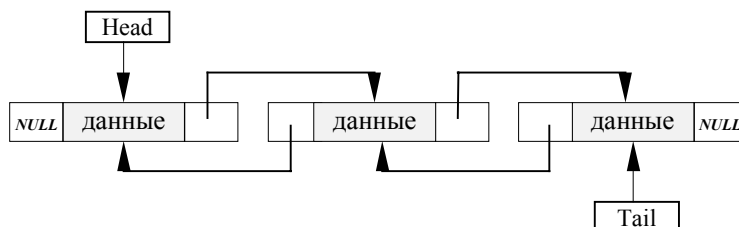
если не нашли, выход

Барьеры

Вы заметили, что для рассмотренного варианта списка требуется отдельно обрабатывать граничные случаи: добавление в начало, добавление в конец, удаление одного из крайних элементов. Можно значительно упростить приведенные выше процедуры, если установить два барьера - фиктивные первый и последний элементы. Таким образом, в списке всегда есть хотя бы два элемента-барьера, а все рабочие узлы находятся между ними.

Двусвязный список

Многие проблемы, присущие односвязным спискам, вызваны тем, что в них невозможно перейти к предыдущему элементу. Возникает естественная идея - хранить в памяти ссылку не только на следующий, но и на предыдущий элемент списка. Для доступа к списку используется не одна переменная-указатель, а две - ссылка на «голову» списка (**Head**) и на «хвост» - последний элемент (**Tail**).



Каждый узел содержит (кроме полезных данных) также ссылку на **следующий** за ним узел (поле **next**) и предыдущий (поле **prev**). Поле **next** у последнего элемента и поле **prev** у первого содержат **NULL**. Узел объявляется так:

```

struct Node {
    char    word[40];
    int     count;
    Node    *next, *prev;
};
typedef Node *PNode;

```

область данных

ссылки на соседние узлы

тип "указатель на узел"

В дальнейшем мы будем считать, что указатель **Head** указывает на начало списка, а указатель **Tail** - на конец списка:

```
PNode Head, Tail;
```

Для пустого списка оба указателя равны **NULL**.



Операции с двусвязным списком

Основными операциями являются добавление элемента в список и исключение из списка. При этих операциях важно правильно работать со ссылками на предыдущие элементы узлов.

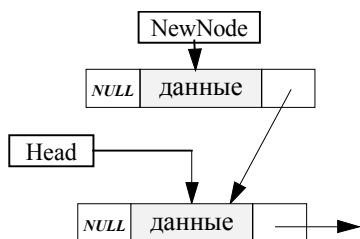


Добавление узла в начало списка

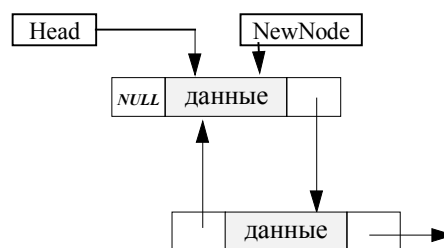
При добавлении нового узла **NewNode** в начало списка надо

- 1) установить ссылку **next** узла **NewNode** на голову существующего списка и его ссылку **prev** в **NULL**;
- 2) установить ссылку **prev** бывшего первого узла (если он существовал) на **NewNode**;
- 3) установить голову списка на новый узел;
- 4) если в списке не было ни одного элемента, хвост списка также устанавливается на новый узел.

1-2)



3-4)



По такой схеме работает следующая процедура:

```
void AddFirst(PNode NewNode)
{
    NewNode->next = Head;
    NewNode->prev = NULL;
    if ( Head ) Head->prev = NewNode;
    Head = NewNode;
    if ( ! Tail ) Tail = Head;
}
```

это первый элемент в списке



Добавление узла в конец списка

Благодаря симметрии добавление нового узла **NewNode** в конец списка проходит совершенно аналогично, в процедуре надо везде заменить **Head** на **Tail** и наоборот, а также поменять **prev** и **next**.



Проход по списку

Проход по двусвязному списку может выполняться в двух направлениях - от головы к хвосту (как для односвязного) или от хвоста к голове.

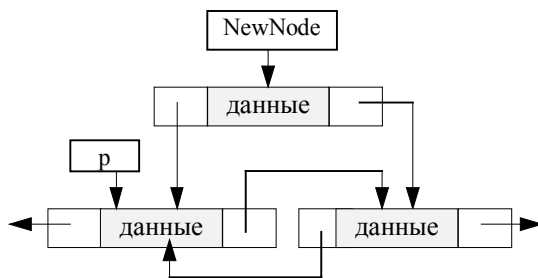


Добавление узла после заданного

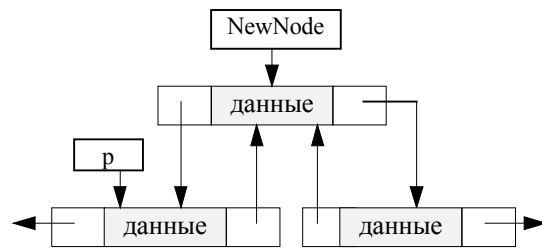
Дан адрес **NewNode** нового узла и адрес **p** одного из существующих узлов в списке. Требуется вставить в список новый узел после данного. Если данный узел является последним, то операция сводится к добавлению в конец списка (см. выше). Если узел **p** - не последний, то операция вставки выполняется в два этапа:

- 1) установить ссылки нового узла на следующий за данным (**next**) и предшествующий ему (**prev**);
- 2) установить ссылки соседних узлов так, чтобы включить **NewNode** в список.

1)



2)



Такой метод реализует приведенная ниже процедура (она учитывает также возможность вставки элемента в конец списка - именно для этого в параметрах передаются ссылки на голову и хвост списка):

```
void    AddAfter (PNode p, PNode NewNode)
{
    if ( !p->next )
        AddLast (Head, Tail, NewNode);
    else {
        NewNode->next = p->next;
        NewNode->prev = p;
        p->next->prev = NewNode;
        p->next = NewNode;
    }
}
```

вставка в конец списка

меняем ссылки нового узла

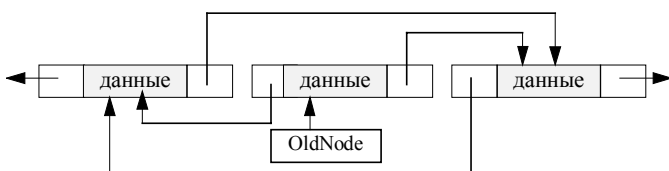
меняем ссылки соседних узлов

Эта операция выполняется почти также с учетом симметричности структуры.

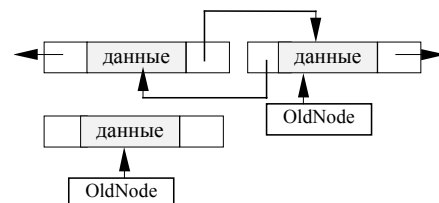
Удаление узла

Эта процедура также требует ссылки на голову и хвост списка, поскольку они могут измениться при удалении крайнего элемента списка. На первом этапе устанавливаются ссылки соседних узлов (если они есть) так, как если бы удаляемого узла не было бы. Затем узел удаляется и память, которую он занимает, освобождается. Эти этапы показаны на рисунке внизу. Отдельно проверяется, не является ли удаляемый узел первым или последним узлом списка.

1)



2)



```
void Delete(PNode &Head, PNode &Tail, PNode OldNode)
{
    if (Head == OldNode) {
        Head = OldNode->next;
        if ( Head )
            Head->prev = NULL;
        else Tail = NULL;
    }
    else {
        OldNode->prev->next = OldNode->next;
        if ( OldNode->next )
            OldNode->next->prev = OldNode->prev;
        else Tail = NULL;
    }
    delete OldNode;
}
```

удаляется первый

удалили единственный элемент

удалили последний элемент



Циклические списки

Иногда список (односвязный или двусвязный) замыкают в кольцо, то есть указатель **next** последнего элемента указывает на первый элемент, и (для двусвязных списков) указатель **prev** первого элемента указывает на последний. В таких списках понятие "хвоста" списка не имеет смысла, для работы с ним надо использовать указатель на "голову", причем "головой" можно считать любой элемент.

2. Стеки, очереди, деки

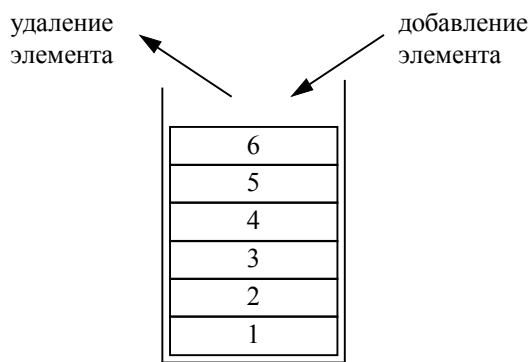
Структуры, перечисленные в заголовке раздела, представляют собой списки, для которых разрешены только операции вставки и удаления первого и/или последнего элемента.



Стек

Стек - это упорядоченный набор элементов, в котором добавление новых и удаление существующих элементов допустимо только с одного конца, который называется **вершиной стека**.

Стек называют структурой типа **LIFO** (*Last In - First Out*) - последним пришел, первым ушел. Моделью стека может служить стопка с подносами, уложенными один на другой - чтобы достать какой-то поднос надо снять все подносы, которые лежат на нем, а положить новый поднос можно только сверху всей стопки. На рисунке показан стек, содержащий 6 элементов.



В современных компьютерах стек используется для

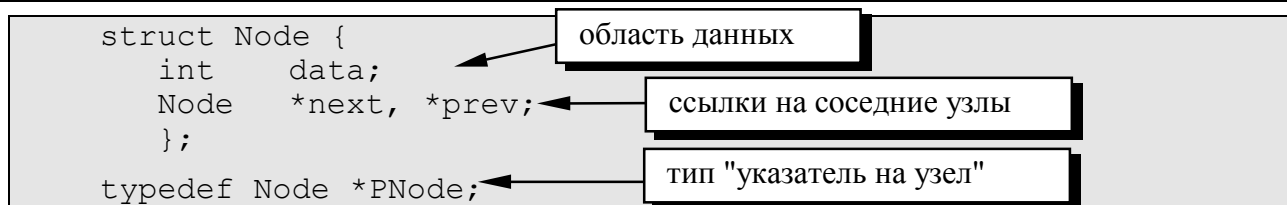
- размещения локальных переменных
- размещения параметров процедуры или функции
- сохранения адреса возврата (по какому адресу надо вернуться из процедуры)
- временного хранения данных, особенно при программировании на Ассемблере

На стек выделяется ограниченная область памяти (в языке Си стандартно 4 Кб). При каждом вызове процедуры в стек добавляются новые элементы (параметры, локальные переменные, адрес возврата). Поэтому если вложенных вызовов будет много, стек переполнится. Очень опасной в отношении переполнения стека является **рекурсия**, поскольку она как раз и предполагает вложенные вызовы одной и той же процедуры или функции. При ошибке в программе рекурсия может стать бесконечной, кроме того, стек может переполниться, если вложенных вызовов будет слишком много.



Реализация стека в языке Си

Рассмотрим пример стека, в котором хранятся целые числа (это простейший вариант, элементом стека могут быть любые типы данных или структур, так же, как и для списка). Для удобства будем реализовывать стек на основе двусвязного списка. Объявление узла списка и указателя на узел будет выглядеть так:



Для удобства чтобы не работать с отдельными указателями на хвост и голову списка, объявим структуру, в которой будет храниться вся информация о стеке:

```

struct Stack {
    PNode Head, Tail;
};

```

В самом начале надо записать в обе ссылки стека **NULL**.

Для стека определены две основные операции:

Добавление элемента на вершину стека

Фактически это добавление нового элемента в начало двусвязного списка. Эта процедура уже была написана ранее, теперь ее придется немного переделать, чтобы работать не с отдельными указателями, а со структурой типа **Stack**. В параметрах процедуры указывается не новый узел, а только **данные** для этого узла, то есть целое число. Память под новый узел выделяется в процедуре, то есть, скрыта от нас и снижает вероятность ошибки.

```

void Push ( Stack &S, int i )
{
    PNode NewNode;
    NewNode = new Node;
    NewNode->data = i;
    NewNode->next = S.Head;
    NewNode->prev = NULL;

    if ( S.Head ) S.Head->prev = NewNode;
    S.Head = NewNode;

    if ( ! S.Tail ) S.Tail = S.Head;
}

```

Получение верхнего элемента с вершины стека

Получение верхнего элемента и удаление его с вершины стека. Для этого надо написать функцию, которая удаляет верхний элемент и возвращает его данные.

```

int Pop ( Stack &S )
{
    PNode TopNode = S.Head;
    int i;

    if ( ! TopNode ) return 0;
    i = TopNode->data;

    S.Head = TopNode->next;

    if ( S.Head ) S.Head->prev = NULL;
    else          S.Tail = NULL;

    delete TopNode;
    return i;
}

```



Системный стек в программах

В языке Си на системный стек по умолчанию отводится **4 Кб** памяти. Это очень немного и довольно частой ошибкой в больших программах является **переполнение стека** - выход за границы отведенной памяти. Чтобы сразу получить сообщение о переполнении, надо включить опцию **Test stack overflow** компилятора. В этом случае при каждом новом вызове процедур программа будет проверять, достаточно ли места в стеке. При переполнении стека происходит аварийный выход. Если в самом деле надо расширить область стека, в начале основного модуля программы напишите

```
extern unsigned _stklen = 10000;
```

расширить стек до 10000 байт

Дело в том, что размер стека определяется системной глобальной переменной **_stklen**, которую мы и изменили.

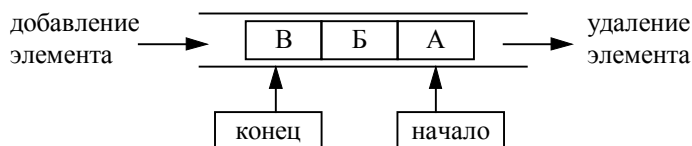
При программировании для **Windows** размер стека по умолчанию равен **1 Мб**, этого хватает для всех стандартных приложений.



Очередь

Очередь - это упорядоченный набор элементов, в котором добавление новых элементов допустимо с одного конца (он называется **начало очереди**), а удаление существующих элементов - только с другого конца, который называется **концом очереди**.

Хорошо знакомой моделью является очередь в магазине. Очередь называют структурой типа **FIFO** (*First In - First Out*) - первым пришел, первым ушел. На рисунке изображена очередь из 3-х элементов.



Наиболее известные примеры применения очередей в программировании - очередь событий системы **Windows** и ей подобных. Очереди используются также для моделирования в **задачах массового обслуживания** (например, обслуживания клиентов в банке).



Дек

Дек (deque) - это упорядоченный набор элементов, в котором добавление новых и удаление существующих элементов допустимо с любого конца.

Дек может быть реализован на основе стека, рассмотренного выше. Остается добавить только две функции - добавление нового элемента в конец (**PushTail**) и получение последнего элемента с его удалением из дека (функция **PopTail**). Они легко получаются из уже написанных **Push** и **Pop** путем замены **Head** на **Tail** и **prev** на **next**, и наоборот.

```
void      PushTail ( Stack &S, int i )
{
    PNode NewNode = new Node;
    NewNode->data = i;

    NewNode->prev = S.Tail;
    NewNode->next = NULL;
    if ( S.Tail ) S.Tail->next = NewNode;
    S.Tail = NewNode;

    if ( ! S.Head ) S.Head = S.Tail;
}
```

Правильнее было бы назвать структуру по-другому, поскольку теперь она обладает не только свойствами стека. Она позволяет моделировать дек, очередь или стек, для каждого типа списков допустимы свои функции. Так, для стека разрешены Push и Pop, а для очереди - Push и PopTail или PushTail и Pop (по выбору). Для полного дека доступны все четыре функции.

```
int PopTail ( Stack &S )
{
    PNode LastNode = S.Tail;
    int i;

    if ( ! LastNode ) return 0;

    i = LastNode->data;
    S.Tail = LastNode->prev;

    if ( S.Tail ) S.Tail->next = NULL;
    else          S.Head = NULL;

    delete LastNode;
    return i;
}
```

3. Деревья



Что такое деревья?

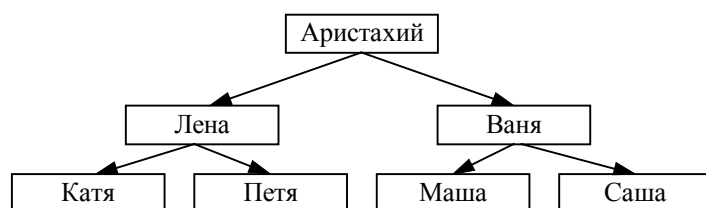


Основные понятия

Дерево - это совокупность **узлов (вершин)** и соединяющих их направленных **ребер (дуг)**, причем в каждый узел (за исключением одного - **корня**) ведет ровно одна дуга.

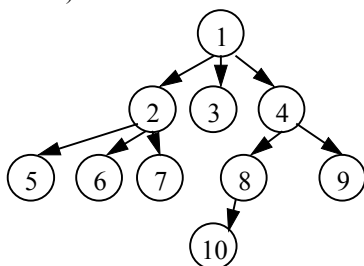
Корень - это начальный узел дерева, в который не ведет ни одной дуги.

Примером может служить **генеалогическое дерево** - в корне дерева находитесь вы сами, от вас идет две дуги к родителям, от каждого из родителей - две дуги к их родителям и т.д.

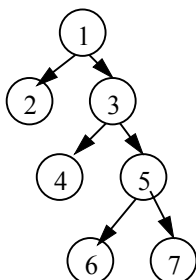


Например, на рисунке структуры а) и б) являются деревьями, а в) и г) - нет.

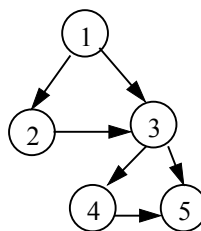
а)



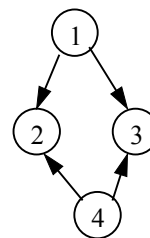
б)



в)



г)



Предком для узла x называется узел дерева, из которого существует путь (по стрелкам) в узел x .

Потомком узла x называется узел дерева, в который существует путь (по стрелкам) из узла x .

Родителем для узла x называется узел дерева, из которого существует непосредственная дуга в узел x .

Сыном узла x называется узел дерева, в который существует непосредственная дуга из узла x .

Уровнем узла x называется длина пути (количество дуг) от корня к данному узлу. Считается, что корень находится на уровне 0.

Листом дерева называется узел, не имеющий потомков.

Внутренней вершиной называется узел, имеющий потомков.

Высотой дерева называется максимальный уровень листа дерева.

Упорядоченным деревом называется дерево, все вершины которого упорядочены (то есть имеет значение последовательность перечисления потомков каждого узла).

Например, два упорядоченных дерева на рисунке ниже - разные.



Рекурсивное определение

Дерево представляет собой типичную рекурсивную структуру (определяемую через саму себя). Как и любое рекурсивное определение, определение дерева состоит из двух частей - первая определяет условие окончания рекурсии, а второе - механизм ее использования.

- 1) пустая структура является деревом
- 2) дерево - это корень и несколько связанных с ним деревьев (поддеревьев)

Таким образом, размер памяти, необходимый для хранения дерева, заранее неизвестен, потому что неизвестно, сколько узлов будет в него входить.

Двоичные деревья

На практике используются главным образом деревья особого вида, называемые **двоичными** (бинарными).

Двоичным деревом называется дерево, каждый узел которого имеет не более двух сыновей.

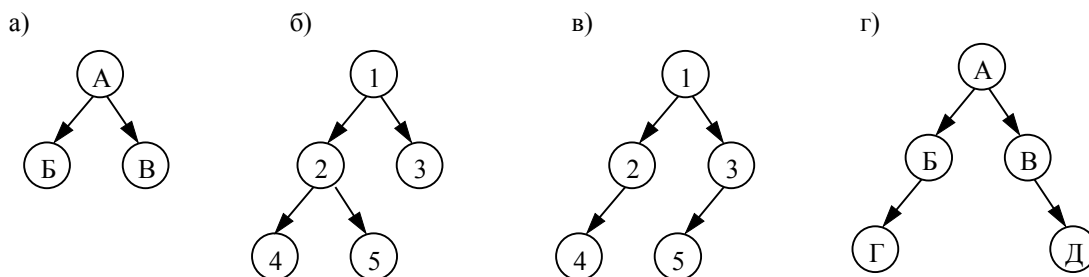
Можно определить двоичное дерево и рекурсивно:

- 1) пустая структура является двоичным деревом
- 2) дерево - это корень и два связанных с ним двоичных дерева, которые называют **левым** и **правым** поддеревом

Двоичные деревья **упорядочены**, то есть различают левое и правое поддерево. Типичным примером двоичного дерева является генеалогическое дерево (родословная). В других случаях двоичные деревья используются тогда, когда на каждом этапе некоторого процесса надо принять одно решение из двух возможных. В дальнейшем мы будем рассматривать только двоичные деревья.

Строго двоичным деревом называется дерево, у которого каждая внутренняя вершина имеет непустые левое и правое поддерево.

Это означает, что в строго двоичном дереве нет вершин, у которых есть только одно поддерево. На рисунке даны деревья а) и б) являются строго двоичными, а в) и г) - нет.



Полным двоичным деревом называется дерево, у которого все листья находятся на одном уровне и каждая внутренняя вершина имеет непустые левое и правое поддеревья.

На рисунке выше только дерево а) является полным двоичным деревом.



Реализация деревьев в языке Си



Описание вершины

Вершина дерева, как и узел любой динамической структуры, имеет две группы данных: полезную информацию и ссылки на узлы, связанные с ним. Для двоичного дерева таких ссылок будет две – ссылка на **левого сына** и ссылка на **правого сына**. В результате получаем структуру, описывающую вершину (предполагая, что полезными данными для каждой вершины является одно целое число):

```
struct Node {
    int Key;
    Node *Left, *Right;
};

typedef Node *PNode;
```

← полезные данные (ключ)
← ссылки на сыновей
← тип "указатель на вершину"



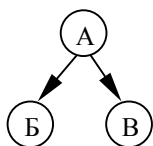
Идеально сбалансированные деревья

Для большинства практических задач наиболее интересны такие деревья, которые имеют минимально возможную высоту при заданном количестве вершин n . Очевидно, что минимальная высота достигается тогда, когда на каждом уровне (кроме, возможно, последнего) будет максимально возможное число вершин.

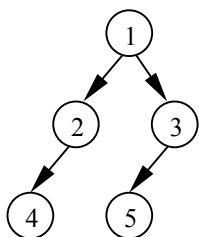
Дерево называется **идеально сбалансированным**, если число вершин в его левом и правом поддеревьях отличается не более чем на 1.

На рисунке деревья а) и б) являются сбалансированными, а деревья в) и г) - нет.

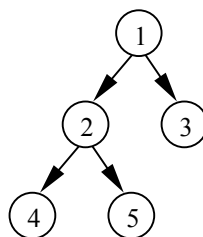
а)



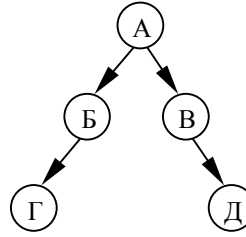
б)



в)



г)



Построение идеально сбалансированных деревьев

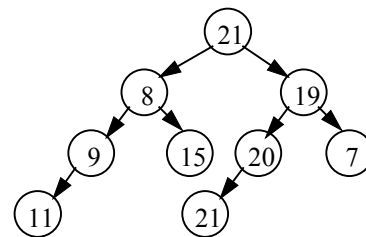
Предположим, что задано n чисел (их количество заранее известно). Требуется построить из них идеально сбалансированное дерево. Алгоритм решения этой задачи предельно прост:

1. Взять одну вершину в качестве корня и записать в нее первое нерассмотренное число.
2. Построить этим же способом идеально сбалансированное левое поддерево из $n_1 = n/2$ вершин (деление нацело!)
3. Построить этим же способом идеально сбалансированное правое поддерево из $n_2 = n - n_1 - 1$ вершин

Заметим, что по построению левое поддереву всегда будет содержать столько же вершин, сколько правое поддереву, или на 1 больше. Для массива данных

21, 8, 9, 11, 15, 19, 20, 21, 7

по этом алгоритму строится идеально сбалансированное дерево, показанное на рисунке справа.



Как будет выглядеть эта программа на языке Си? Надо сначала разобраться, что означает "взять одну вершину в качестве корня и записать туда первое нерассмотренное число". Поскольку вершины должны создаваться динамически, надо **выделить память** под вершину и записать в поле данных нужное число. Затем из оставшихся чисел построить левое и правое поддеревья.

В основной программе нам надо объявить указатель на корень нового дерева, задать массив данных (в принципе можно читать данные из файла) и вызвать функцию, возвращающую указатель на построенное сбалансированное дерево.

```
int data[] = { 21, 8, 9, 11, 15, 19, 20, 21, 7 };
```

```
PNode Tree;
```

```
n = sizeof(data) / sizeof(int) - 1;
```

```
Tree = MakeTree (data, 0, n);
```

указатель на корень дерева

размер массива

использовать **n** элементов

начать с элемента 0

Сама функция MakeTree принимает три параметра: массив данных, номер первого неиспользованного элемента и количество элементов в новом дереве. Возвращает она указатель на новое дерево (типа PNode).

```
PNode MakeTree (int data[], int &from, int n)
```

```
{
```

```
PNode Tree;
```

```
int n1, n2;
```

```
if ( n == 0 ) return NULL;
```

```
Tree = new Node;
```

```
Tree->Key = data[from++];
```

```
n1 = n / 2;
```

```
n2 = n - n1 - 1;
```

```
Tree->Left = MakeTree(data, from, n1);
```

```
Tree->Right = MakeTree(data, from, n2);
```

```
return Tree;
```

```
}
```

передача по ссылке

ограничение рекурсии

выделить память под вершину

записать данные и перейти к следующему элементу

размеры левого и правого поддеревьев

Номер первого невыбранного элемента (параметр **from**) надо передавать **по ссылке** (объявлять со знаком **&**), потому что он изменяется при каждом новом рекурсивном вызове. Другой вариант - сделать массив **data** и переменную **from** глобальными и исключить их из списка параметров. Однако при этом теряется гибкость процедуры - очень сложно будет в одной программе строить несколько разных деревьев с разными данными, поскольку процедура будет жестко привязана к глобальным переменным.



Обход дерева

Одной из необходимых операций при работе с деревьями является **обход дерева**, во время которого надо посетить каждый узел по одному разу и (возможно) вывести информацию, содержащуюся в вершинах.

Пусть в результате обхода надо напечатать значения поля данных всех вершин в определенном порядке. Существуют **три варианта обхода**:

1. **Просмотр в ширину (сверху вниз)**, при котором сначала посещается корень (выводится информация о нем), затем левое поддерево, а затем - правое. Такой обход называют обходом типа КЛП (*корень - левое - правое*).
2. **Просмотр в симметричном порядке (слева направо)**, при котором сначала посещается левое поддерево, затем корень, а затем - правое. Такой обход называют обходом типа ЛКП (*левое - корень - правое*).
3. **Просмотр снизу вверх**, при котором сначала посещается левое поддерево, затем правое, а затем - корень. Такой обход называют обходом типа ЛПК (*левое - правое - корень*).

Для примера ниже дана рекурсивная процедура просмотра дерева слева направо. Обратите внимание, что поскольку дерево является рекурсивной структурой данных, при работе с ним естественно широко применять рекурсию.

```
void PrintLKP(PNode Tree)
{
    if ( ! Tree ) return;
    PrintLKP(Tree->Left);
    printf("%d ", Tree->Key);
    PrintLKP(Tree->Right);
}
```

Остальные варианты обхода программируются аналогично.



Сортировка и поиск с помощью дерева



Как быстрее искать?

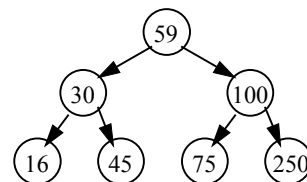
Деревья очень удобны для поиска в них информации. Предположим, что существует массив данных и с каждым элементом связан ключ - число, по которому выполняется поиск. Пусть ключи для элементов таковы:

59, 100, 75, 30, 16, 45, 250

Для этих данных нам надо много раз проверять, присутствует ли среди ключей заданный ключ **x**, и если присутствует - вывести всю связанную с этим элементом информацию.

Если данные организованы в виде неотсортированного массива, то для поиска в худшем случае надо сделать **n** сравнений элементов (сравнивая последовательно с каждым элементом пока не найдется нужный или пока не закончится массив).

Теперь предположим, что данные организованы в виде дерева, показанного на рисунке. Такое дерево обладает следующим важным свойством:



Значения ключей всех вершин левого поддерева вершины **x** меньше ключа **x**, а значения ключей всех вершин правого поддерева **x** больше или равно ключу вершины **x**.

Для поиска нужного элемента в таком дереве требуется не более 3 сравнений вместо 7 при поиске в списке или массиве, то есть поиск проходит значительно быстрее.

Сортировка с помощью дерева

Как же, имея массив данных, построить такое дерево?

1. Сравнить ключ очередного элемента массива с ключом корня.
2. Если ключ нового элемента меньше, включить его в левое поддерево, если больше или равен, то в правое.
3. Если текущее дерево пустое, создать новую вершину и включить в дерево.

Программа, приведенная ниже, реализует этот алгоритм:

```
void AddToTree (PNode &Tree, int data)
{
    if ( ! Tree ) {
        Tree = new Node;
        Tree->Key = data;
        Tree->Left = NULL;
        Tree->Right = NULL;
        return;
    }

    if ( data < Tree->Key )
        AddToTree ( Tree->Left, data );
    else AddToTree ( Tree->Right, data );
}
```

указатель на корень, он
может измениться

создать новый узел

добавить в левое или
правое поддерево

Важно, что указатель на корень дерева надо передавать по ссылке, так как он может измениться при создании новой вершины. Чтобы получить все ключи по возрастанию, надо пройти дерево **слева направо**, распечатывая ключи вершин.

Надо заметить, что в результате работы этого алгоритма не всегда получается дерево минимальной высоты – все зависит от порядка выбора элементов. Для оптимизации поиска используют так называемые **сбалансированные** или **АВЛ-деревья**¹ (но не идеально сбалансированные!!!) деревья, у которых для любой вершины высоты левого и правого поддеревьев отличаются не более, чем на 1. Добавление в них нового элемента иногда сопровождается некоторой перестройкой дерева.

Поиск одинаковых элементов

Приведенный алгоритм можно модифицировать так, чтобы быстро искать одинаковые элементы в массиве чисел. Один из способов решения этой задачи - перебрать все элементы массива и сравнить каждый со всеми остальными. Однако для этого требуется очень большое число сравнений. С помощью двоичного дерева можно значительно ускорить поиск. Для этого надо в структуру вершины включить еще одно поле - счетчик найденных дубликатов **count**.

```
struct Node {
    int    Key;
    int    Count;
    Node  *Left, *Right;
};
```

счетчик дубликатов

¹ Сбалансированные деревья называют так в честь изобретателей этого метода Г.М. Адельсона-Вельского и Е.М. Ландиса.

При создании узла в счетчик записывается единица (найден один элемент). Поиск дубликатов происходит по следующему алгоритму:

1. Сравнить ключ очередного элемента массива с ключом корня.
2. Если ключ нового элемента равен ключу корня, то увеличить счетчик корня и стоп.
3. Если ключ нового элемента меньше, включить его в левое поддерево, если больше или равен - в правое.
4. Если текущее дерево пустое, создать новую вершину (со значением счетчика 1) и включить в дерево.



Поиск по дереву

Теперь, когда дерево сортировки построено, очень легко искать элемент с заданным ключом. Сначала проверяем ключ корня, если он равен искомому, то нашли. Если он меньше искомого, ищем в левом поддереве корня, если больше - то в правом. Приведенная функция возвращает адрес нужной вершины, если поиск успешный, и **NULL**, если требуемый элемент не найден.

```
PNode Search (PNode Tree, int what)
{
    if ( !Tree ) return NULL;
    if ( what == Tree->Key ) return Tree;
    if ( what < Tree->Key )
        return Search ( Tree->Left, what );
    else return Search ( Tree->Right, what );
}
```

ключ не найден

ключ найден

искать в поддеревьях



Разбор арифметического выражения

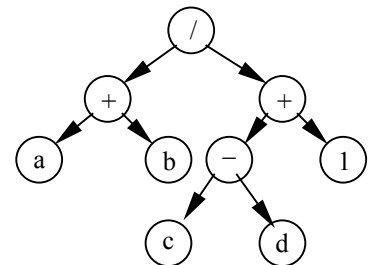


Дерево для арифметического выражения

Вы задумывались над тем, как транслятор обрабатывает и выполняет арифметические и логические выражения, которые он встречает в программе? Один из вариантов - представить это выражение в виде двоичного дерева. Например, выражению

$$(a + b) / (c - d + 1)$$

соответствует дерево, показанное на рисунке слева. Листья содержат числа и имена переменных (операндов), а внутренние вершины и корень - арифметические действия и вызовы функций. Вычисляется такое выражение снизу, начиная с листьев. Как видим, скобки отсутствуют и дерево полностью определяет порядок выполнения операций.



Формы записи арифметического выражения

Теперь посмотрим, что получается при прохождении таких двоичных деревьев. Прохождение дерева в ширину (корень - левое - правое) дает

$$/ + a b + - c d 1$$

то есть знак операции (корень) предшествует своим операндам. Такая форма записи арифметических выражений называется **префиксной**. Проход в прямом порядке (левое - корень - правое) дает **инфиксную форму**, которая совпадает с обычной записью, но без учета скобок:

$$a + b / c - d + 1$$

Поскольку скобок нет, правильный порядок операций невозможно восстановить по инфиксной записи. В трансляторах широко используется **постфиксная запись** выражений, которая получается в результате обхода **снизу вверх** (левое - правое - корень). В ней знак операции стоит **после** обоих операндов:

$$a \ b \ + \ c \ d \ - \ 1 \ /$$

Порядок выполнения такого выражения однозначно определяется следующим алгоритмом, который использует стек:

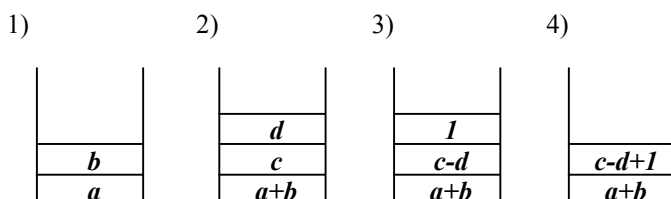
Пока в постфиксной записи есть невыбранные элементы, делай:

1. Взять очередной элемент.
2. Если это операнд (не знак операции), то записать его в стек.
3. Если это знак операции, то
 - выбрать из стека второй операнд
 - выбрать из стека первый операнд
 - выполнить операцию с этими данными и результат записать в стек

Проиллюстрируем на примере вычисление выражения в постфиксной форме

$$a \ b \ + \ c \ d \ - \ 1 \ /$$

Согласно алгоритму, сначала запишем в стек **a**, а затем **b** (рисунок 1).



Результат выполнения операции **a+b** запишем обратно в стек, а сверху - выбранные из входного потока значения переменных **c** и **d** (рисунок 2). Дальнейшее развитие событий показано на рисунках 3 и 4. Выполнение последней операции (деления) для стека на рисунке 4 дает искомый результат.



Алгоритм построения дерева

Пусть задано арифметическое выражение. Надо построить для него дерево синтаксического разбора и различные форму записи.

Чтобы не слишком усложнять задачу, рассмотрим самый простой вариант, введя следующие упрощения:

1. В выражении могут присутствовать целые числа, имена переменных и знаки операций **+**, **-**, *****, **/**.
2. Запрещается использование вызовов функций, скобок, унарных знаков плюс и минус (например, запрещено выражение **-a+5**, вместо него надо писать **0-a+5**).
3. Предполагается, что выражение записано верно, то есть не делается проверки на правильность.
4. Выражение уже разбито на отдельные элементы трех типов (числа, переменные, знаки операций), записанные в массив символьных строк **Term** размером **N**.

Вспомним, что порядок выполнения операций в выражении определяется **приоритетом операций** - первыми выполняются операции с более высоким приоритетом. Например, умножение и деление выполняются раньше, чем сложение и вычитание.

Будем строить дерево для элементов массива с номерами от **first** до **last** (полное дерево дает применение этого алгоритма ко всему массиву, то есть при **first=0** и **last=N-1**). В словесном виде алгоритм выглядит так:

1. Если **first=last** (остался один элемент – переменная или число), то создать новый узел и записать в него этот элемент. Иначе...
2. Среди элементов от **first** до **last** включительно найти **последнюю** операцию с наименьшим приоритетом (пусть найденный элемент имеет номер **k**).
3. Создать новый узел (корень) и записать в него знак операции **Term[k]**.
4. Рекурсивно применить этот алгоритм два раза:
 - построить левое поддерево, разобрав выражение из элементов массива с номерами от **first** до **k-1**
 - построить правое поддерево, разобрав выражение из элементов массива с номерами от **k+1** до **last**

Чтобы написать программу на Си, надо определить функцию, возвращающую приоритет операции, которая ей передана. Определим приоритет 1 для сложения и вычитания и приоритет 2 для умножения и деления.

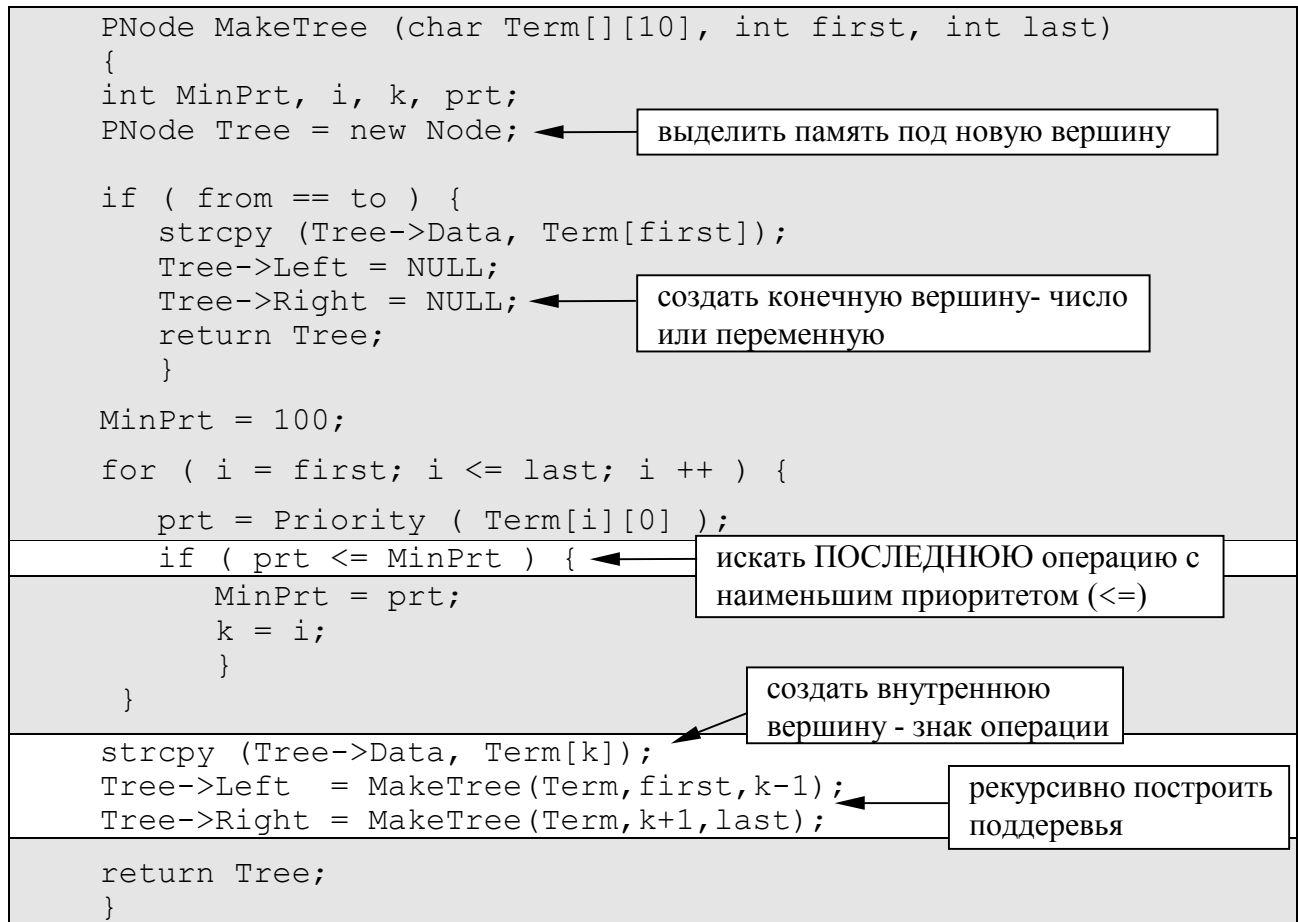
```
int Priority ( char c )
{
    switch ( c ) {
        case '+':
        case '-':
            return 1;

        case '*':
        case '/':
            return 2;

        }
    return 100;
}
```

← это не операция, пропускаем

Приведенная ниже программа строит требуемое дерево, используя эту функцию. Обратите внимание, что при сравнении приоритета текущей операции с минимальным предыдущим используется условие \leq . За счет этого мы ищем именно **последнюю** операцию с минимальным приоритетом, то есть, операцию, которая будет выполняться самой последней. Если бы мы использовали знак $<$, то нашли бы **первую** операцию с наименьшим приоритетом и дерево было бы построено неверно.



Теперь обход этого дерева разными способами дает различные формы представления соответствующего арифметического выражения.

Разбор выражения со скобками

Немного усложним задачу и разрешим использовать в выражении скобки одного вида (допустим, круглые). Тогда при поиске в заданном диапазоне операции с минимальным приоритетом не надо брать во внимание выражения в скобках (они выделены на рисунке).

$$a + ((b + c) * 5 + 3) * 7$$

Самый простой способ добиться этого эффекта – ввести счетчик открытых скобок **nest**. В начале он равен нулю, с каждой найденной открывающей скобкой будем увеличивать его на 1, а с каждой закрывающей - уменьшать на 1. Рассматриваются только те операции, которые найдены при **nest=0**, то есть расположены вне скобок.

Если же ни одной такой операции не найдено, то мы имеем выражение, ограниченной скобками, поэтому надо вызвать процедуру рекурсивно для диапазона **from+1..last-1** (напомним, что мы предполагаем, что выражение корректно). Для сокращения записи показаны только те части процедуры, которые изменяются:

```

PNode MakeTree (char Term[][10], int first, int last)
{
    char c;
    int MinPrt, i, k, prt;
    int nest = 0;
    PNode Tree = new Node;

    ...
    MinPrt = 100;
    for ( i = first; i <= last; i ++ ) {
        c = Term[i][0];
        if ( c == '(' ) { nest ++; continue; }
        if ( c == ')' ) { nest --; continue; }
        if ( nest > 0 ) continue;
        prt = Priority ( c );
        if ( prt <= MinPrt ) {
            MinPrt = prt;
            k = i;
        }
    }
    if ( MinPrt == 100 &&
        Term[first][0] == '(' && Term[last][0] == ')' )
        return MakeTree(Term, first+1, last-1);

    ...
    return Tree;
}

```

счетчик вложенности
скобок

обработать скобки

пропускаем все, что в скобках

все выражение взято в скобки

Упрощение выражения с помощью дерева

Некоторые выражения можно сразу значительно упростить, используя очевидные тождества, верные для любого x :

$0 + x = x$	$x + 0 = x$	$0 * x = 0$
$0 - x = -x$	$x - 0 = x$	$1 * x = x$

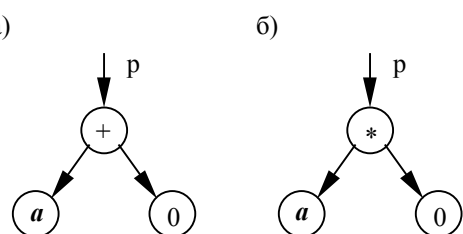
Пусть, например, мы нашли такую структуру, как показано на рисунке *а*. Значение всего первого выражения равно *а*, поэтому нам надо сделать следующее: указатель *р* поставить на вершину *а*, а две ненужные вершины удалить из памяти.

В случае *б*) аналогично надо указатель *р* переставить на вершину со значением 0. при этом надо учесть, что второй узел (со значением *а*) может иметь потомков, которых также надо корректно удалить. Это делается рекурсивно:

```

void DeleteNode ( PNode Tree )
{
    if ( Tree == NULL ) return;
    DeleteNode ( Tree->Left );
    DeleteNode ( Tree->Right );
    delete Tree;
}

```



Кроме того, если оба сына какой-то вершины являются листьями и содержат числа, такое выражение можно сразу посчитать, также удалив два ненужных узла. Один из вариантов

реализации этой операции приведен ниже. Здесь используется функция **IsNumber**, которая возвращает 1, если узел является листом и содержит число, и 0 в противном случае:

<code>int IsNumber (PNode Tree)</code>	
<code>{</code>	
<code>int i = 0;</code>	
<code>if (! Tree) return 0;</code>	пустое дерево
<code>while (Tree->Data[i])</code>	пока не дошли до конца строки
<code>if (! strchr("0123456789", Tree->Data[i++]))</code>	
<code>return 0;</code>	если нашли не цифру, выход
<code>return 1;</code>	
<code>}</code>	

Сама процедура вычисления выражения выглядит так:

<code>void Calculate(PNode Tree)</code>	
<code>{</code>	
<code>int a, b, c = 0;</code>	проверить, можно ли вычислить
<code>if (! Tree ! IsNumber(Tree->Left) </code>	
<code>! IsNumber(Tree->Right)) return;</code>	
<code>a = atoi(Tree->Left->Data);</code>	получить данные от сыновей
<code>b = atoi(Tree->Right->Data);</code>	
<code>switch (Tree->Data[0]) {</code>	
<code>case '+': c = a + b; break;</code>	
<code>case '-': c = a - b; break;</code>	выполнить операцию
<code>case '*': c = a * b; break;</code>	
<code>case '/': c = a / b; break;</code>	
<code>}</code>	
<code>delete Tree->Left;</code>	удалить ненужные поддеревья
<code>delete Tree->Right;</code>	
<code>sprintf(Tree->Data, "%d", c);</code>	обновить вершину
<code>Tree->Left = NULL;</code>	
<code>Tree->Right = NULL;</code>	
<code>}</code>	



Дерево игр

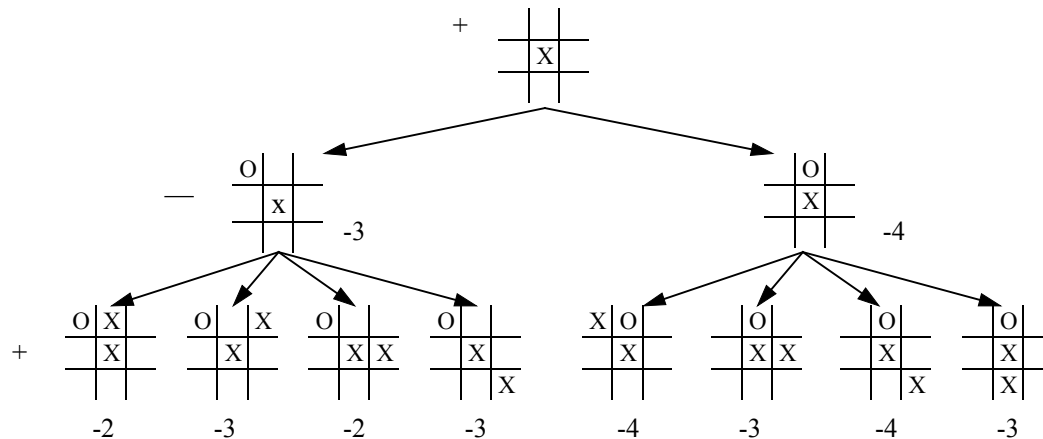
Одно из применений деревьев - игры с компьютером. Рассмотрим самый простой пример - игру в крестики-нолики на поле 3 на 3.

Программа должна анализировать позицию и находить лучший ход. Для этого нужно определить **оценочную функцию**, которая получая позицию на доске и указание, чем играет игрок (крестики или нолики) возвращает число – оценку позиции. Чем она выше, тем более выгодна эта позиция для игрока. Примером такой функции может служить сумма строк, столбцов и диагоналей, которые может занять игрок минус такая же сумма для его противника.

Однако, в этой ситуации программа не ведет просчет вперед и не оценивает позиции, которые могут возникнуть из текущей. Это недостаточно для предсказания исхода игры. Хотя для крестиков-ноликов можно перебрать все варианты и найти выигрышную позицию, большинство игр слишком сложно, чтобы допускать полный перебор.

Выбор хода может быть существенно улучшен, если просматривать на несколько ходов вперед. **Уровнем просмотра** называется число будущих рассматриваемых ходов. Начиная с

любой позиции можно построить дерево возможных позиций, получающихся после каждого хода игры. Для крестиков-ноликов ниже приведено дерево с уровнем просмотра 3 для того случая, когда крестики сделали первый ход в центр доски.



Обозначим игрока, который ходит в корневой позиции (в данном случае - нолики) знаком "плюс", а его соперника - знаком "минус". Попытаемся найти лучший ход для игрока "плюс" в этой позиции. Пусть все варианты следующих ходов были оценены для игрока "плюс". Он должен выбрать такой, в котором оценка максимальная для него.

С другой стороны, как только игрок "плюс" сделает свой ход, игрок "минус" из всех возможных ходов сделает такой, чтобы его оценка с позиции игрока "плюс" была минимальной. Поэтому значение минусового узла для игрока "плюс" равно минимальному из значений сыновей этого узла. Это означает, что на каждом шаге соперники делают наилучшие возможные ходы.

Для того, чтобы выбрать оптимальный ход в корне дерева, надо оценить позицию в его листьях. После этого каждому плюсовому узлу присваивается **максимальное** значение из значений его сыновей, а каждому минусовому - **минимальное**. Такой метод называется методом **минимакса**, так как по мере продвижения вверх используются попеременно функции максимума и минимума. Общая идея метода состоит в том, чтобы выбрать лучший ход на случай худших (для нас) действий противника. Таким образом, лучшим для ноликов в корневой позиции будет ход в угол.

4. Графы



Основные понятия²



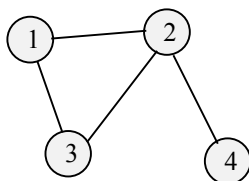
Определения

Во многих жизненных ситуациях старая привычка толкает нас рисовать на бумаге точки, обозначающие людей, города, химические вещества, и показывать линиями (возможно со стрелками) связи между ними. Описанная картинка называется **графом**.

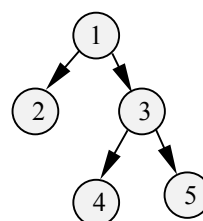
Граф - это совокупность **узлов (вершин)** и соединяющих их **ребер (дуг)**.

Ниже показаны примеры графов

а)



б)



Если дуги имеют направление (вспомните улицы с односторонним движением), то такой граф называется **направленным** или **ориентированным графом (орграфом)**.

Цепью называется последовательность ребер, соединяющих две (возможно не соседние) вершины u и v . В направленном графе такая последовательность ребер называется "**путь**".

Граф называется **связным**, если существует цепь между любой парой вершин. Если граф не связный, то его можно разбить на k связных компонент – он называется **k -связным**.

В практических задачах часто рассматриваются **взвешенные графы**, в которых каждому ребру приписывается **вес** (или длина). Такой граф называют **сетью**.

Циклом называется цепь из какой-нибудь вершины v в нее саму.

Деревом называется граф без циклов.

Полным называется граф, в котором проведены все возможные ребра (для графа, имеющего n вершин таких ребер будет $n(n-1)/2$).



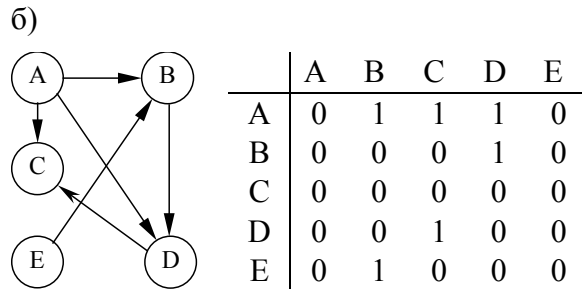
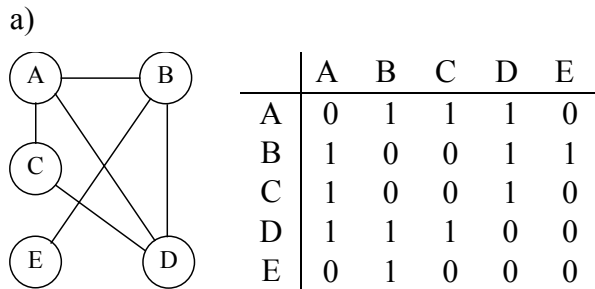
Описание графов

Для описания графов часто используют два типа матриц – **матрицу смежности** (для невзвешенных графов) и **весовую матрицу** (для взвешенных).

Матрица смежности графа с N вершинами – это матрица размером N на N , где каждый элемент с индексами (i, j) является логическим значением и показывает, есть ли дуга из вершины i в вершину j .

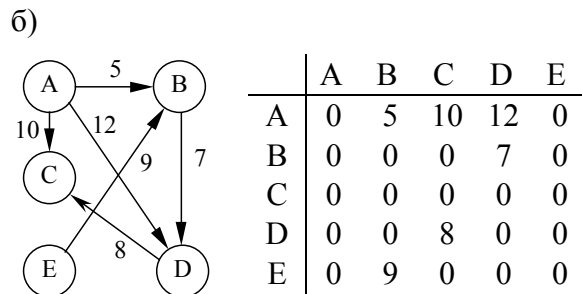
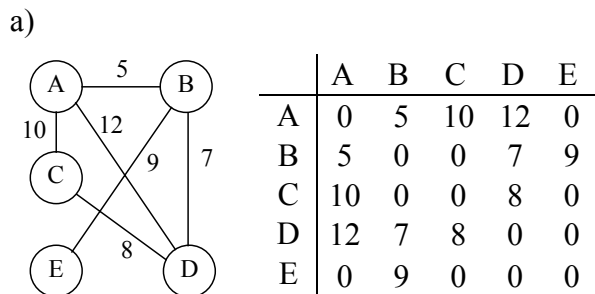
² Этот раздел написан на базе материала книги В.М. Бондарев, В.И. Рублинецкий, Е.Г. Качко. Основы программирования. – Харьков: «Фолио», 1998.

Часто вместо логических значений (истина/ложь) используют целые числа (1/0). Для неориентированных графов матрица смежности всегда симметрична относительно главной диагонали (рисунок а). Для ориентированных графов (рисунок б) это не всегда так, потому что может существовать путь из вершины i в вершину j и не существовать обратного пути.



Для взвешенных графов недостаточно просто указать, есть ли связь между вершинами. Требуется еще хранить в памяти «вес» каждого ребра, например, стоимость проезда или длину пути. Для этого используется **весовая матрица**.

Весовая матрица графа с N вершинами – это матрица размером N на N , где каждый элемент с индексами (i, j) равен «весу» ребра из вершины i в вершину j .



Задача Прима-Краскала



Формулировка задачи

Задача. Дана плоская страна и в ней n городов с известными координатами. Нужно соединить все города телефонной сетью так, чтобы длина телефонных линий была минимальная.

Город будем изображать узлом (точкой). Телефонные линии могут разветвляться только на телефонных станциях, а не в чистом поле. Поскольку требуется линия минимальной общей длины, в ней не будет циклов, потому что иначе можно было бы убрать одно звено цикла и станции по-прежнему были бы связаны. В терминах теории графов эта задача звучит так:

Дан граф с n вершинами; длины ребер заданы матрицей $\{d_{ij}\}$, $i, j=1..n$. Найти набор ребер, соединяющий все вершины графа (он называется **остовным деревом**) и имеющий минимальную длину

Эта задача - одна из тех немногих, для которых известно точное и несложное решение, причем алгоритм предельно прост.



Жадные алгоритмы

Представим себе зимовщика, которому предоставили некоторый запас продуктов на всю зиму. Конечно, он может сначала съесть все самое вкусное - шоколад, мясо и т.п., но за такой подход придется жестоко расплачиваться в конце зимовки, когда останется только соль и маргарин.

Подобным образом, если оптимальное решение строится по шагам, обычно нельзя выбирать на каждом этапе "самое вкусное" - за это придется расплачиваться на последних шагах. Такие алгоритмы называют **жадными**.

Решение

Удивительно, но для непростой задачи Прима-Краскала жадный алгоритм дает точное оптимальное решение. Алгоритм формулируется так:

В цикле **$n-1$** раз выбрать из оставшихся ребер самое короткое ребро, которое не образует цикла с уже выбранными.

Как же проследить, чтобы не было циклов? Оказывается очень просто: в самом начале покрасим все вершины в разные цвета и затем, выбрав очередное ребро между вершинами **i** и **j** , где **i** и **j** имеют разные цвета, перекрасим вершину **j** и все соединенные с ней (то есть имеющие ее цвет) в цвет **i** . Таким образом, при выборе ребер, соединяющих вершины разного цвета, цикл не возникнет никогда, а после **$n-1$** шагов все вершины будут иметь один цвет.

Для записи информации о ребрах введем структуру

```
struct rebro { int i, j; }; ← вершины, которые соединены
```

причем будем считать, что в паре номер первой вершины **i** меньше, чем номер второй **j** .

Приведенная ниже программа действует по следующему «жадному» алгоритму:

1. Покрасить все вершины в разные цвета.
2. Сделать **$n-1$** раз в цикле
 - выбрать ребро **(i, j)** минимальной длины, соединяющее вершины разного цвета;
 - запомнить его в массиве ребер;
 - перекрасить все вершины, имеющие цвет **j** , в цвет **i** .
3. Вывести результат.

```
const    N = 5;

void main()
{
    int    D[N][N], Col[N], i, j,
           k, Dmin, jMin, iMin, col_j;
    rebro Reb[N-1];
    ...           // здесь надо ввести матрицу D

    for ( i = 0; i < N; i ++ )
        Col[i] = i;

    for ( k = 0; k < N-1; k ++ ) {
        Dmin = 30000;

        for ( i = 0; i < N-1; i ++ )
            for ( j = i+1; j < N; j ++ )
                if ( Col[i] != Col[j] && D[i][j] < Dmin ) {
                    Dmin = D[i][j];
                    iMin = i;
                    jMin = j;
                }

        Reb[k].i = iMin;
        Reb[k].j = jMin;
    }
```

покрасить все вершины
в разные цвета

найти самое короткое ребро, не
образующее циклов

запомнить найденное ребро

```

col_j = Col[jMin];
for ( i = 0; i < N; i ++ )
    if ( Col[i] == col_j )
        Col[i] = Col[iMin];
}
... // здесь надо вывести найденные ребра
}

```

перекрасить все вершины, цвет которых совпадает с цветом вершины *jMin*

Дополнительно можно рассчитывать общую длину выбранных ребер, но это не меняет принципиально алгоритм. Этот алгоритм требует памяти порядка n^2 (обозначается $O(n^2)$, то есть при увеличении n в 2 раза объем требуемой памяти увеличивается в 4 раза). Его временная сложность – $O(n^3)$, то есть при увеличении n в 2 раза время вычислений увеличивается в 8 раз (надо просмотреть $O(n^3)$ чисел и сделать это $n-1$ раз).



Кратчайший путь



Формулировка задачи

Задача. Задана сеть дорог между населенными пунктами (часть из них могут иметь одностороннее движение). Требуется найти кратчайший путь между двумя заданными пунктами.

Обычно задачу несколько расширяют и находят сразу кратчайшие пути от заданной вершины ко всем остальным. В терминах теории графов задача выглядит так:

В сети, где часть дорог имеет одностороннее движение, найти кратчайшие пути от заданной вершины ко всем остальным.



Алгоритм Дейкстры

Ниже описывается алгоритм, предложенный Дейкстрой в 1959 г. Дана матрица $\{d_{ij}\}$ длин дуг между вершинами, если дуги между вершинами i и j нет, то $d_{ij}=\infty$. Если сеть образуют n вершин, то для реализации алгоритма требуется три массива длиной n :

1. Массив $\{a_i\}$, в котором $a_i=0$, если вершина i еще не рассмотрена, и $a_i=1$, если вершина i уже рассмотрена.
2. Массив $\{b_i\}$, в котором b_i - текущее кратчайшее расстояние от выбранной стартовой вершины x до вершины i .
3. Массив $\{c_i\}$, в котором c_i - номер предпоследней вершины в текущем кратчайшем пути из выбранной стартовой вершины x до вершины i .

Сам алгоритм состоит из трех этапов: инициализации, основного цикла и вывода результата.

Инициализация

Пусть x - номер выбранной стартовой вершины.

1. Заполним весь массив a значением 0 (пока ни одна вершина не рассмотрена).
2. В i -ый элемент массива b запишем расстояние от вершины x до вершины i (если пути нет, то оно равно ∞ , в программе укажем очень большое число).
3. Заполним весь массив c значением x (пока рассматриваем только прямые пути от x к i).
4. Рассмотрим стартовую вершину: присвоим $a[x]=1$ и $c[x]=0$ (начало всех путей).

Основной цикл

Среди нерассмотренных вершин (для которых $a[i]=0$) найти такую вершину j , что расстояние b_j – минимальное. Рассмотреть ее:

1. Установить $a[j]=1$ (вершина рассмотрена).
2. Сделать для всех вершин k :
 если путь от вершины x к вершине k через j короче, чем уже найденный кратчайший путь (то есть $b_j + d_{jk} < b_k$), то запомнить его: $c[k]=j$ и $b_k = b_j + d_{jk}$

Вывод результата

Можно доказать, что в конце такой процедуры массив b , будет содержать кратчайшие расстояния от стартовой вершины x до вершины i (для всех i). Однако хотелось бы еще получить сам оптимальный путь.

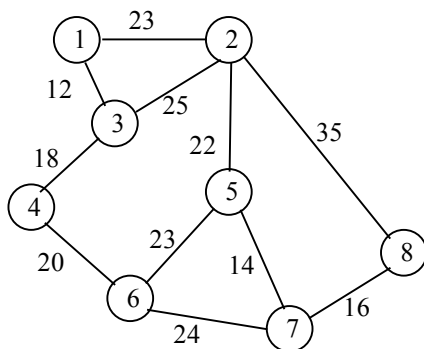
Оказывается, массив c содержит всю необходимую информацию для решения этой задачи. Предположим, что надо вывести оптимальный путь из вершины x в вершину i . По построению предпоследняя вершина в этой цепи имеет номер $z=c[i]$. Теперь надо найти оптимальный путь в вершину z тем же способом. Таким образом, путь "раскручивается" с конца. Когда мы получили $z=0$, путь закончен, мы вернулись в стартовую вершину. В виде алгоритма это можно записать так:

1. Установить $z=i$.
2. Пока $c[z]$ не равно нулю, делай
 - $z=c[z]$
 - вывести z

Для выполнения алгоритма надо n раз просмотреть массив b из n элементов, поэтому он имеет квадратичную сложность $O(n^2)$.

**Пример**

Пусть дана сеть дорог:



Найдем кратчайшие расстояния от вершины 3 до всех остальных вершин. Покажем ход выполнения алгоритма Дейкстры по шагам.

- **Инициализация.** В результате этого шага получаем такие матрицы:

	1	2	3	4	5	6	7	8
a	0	0	1	0	0	0	0	0
b	12	25	0	18	∞	∞	∞	∞
c	3	3	0	3	3	3	3	3

- **Основной цикл.** Последовательно матрицы преобразуются к виду:

1) $\min b_j=12$ для $j=1$

	1	2	3	4	5	6	7	8
<i>a</i>	1	0	1	0	0	0	0	0
<i>b</i>	12	25	0	18	∞	∞	∞	∞
<i>c</i>	3	3	0	3	3	3	3	3

2) $\min b_j=18$ для $j=4$

	1	2	3	4	5	6	7	8
<i>a</i>	1	0	1	1	0	0	0	0
<i>b</i>	12	25	0	18	∞	38	∞	∞
<i>c</i>	3	3	0	3	3	4	3	3

3) $\min b_j=25$ для $j=2$

	1	2	3	4	5	6	7	8
<i>a</i>	1	1	1	1	0	0	0	0
<i>b</i>	12	25	0	18	47	38	∞	60
<i>c</i>	3	3	0	3	2	4	3	2

4) $\min b_j=38$ для $j=6$

	1	2	3	4	5	6	7	8
<i>a</i>	1	1	1	1	0	1	0	0
<i>b</i>	12	25	0	18	47	38	62	60
<i>c</i>	3	3	0	3	2	4	6	2

5) $\min b_j=47$ для $j=5$

	1	2	3	4	5	6	7	8
<i>a</i>	1	1	1	1	1	1	0	0
<i>b</i>	12	25	0	18	47	38	61	60
<i>c</i>	3	3	0	3	2	4	5	2

6) $\min b_j=60$ для $j=8$

	1	2	3	4	5	6	7	8
<i>a</i>	1	1	1	1	1	1	0	1
<i>b</i>	12	25	0	18	47	38	61	60
<i>c</i>	3	3	0	3	2	4	5	2

- **Вывод кратчайшего пути.** Найдем, например, кратчайший путь из вершины 3 в вершину 8. "Раскручивая" массив *c*, получаем $8 \leftarrow 2 \leftarrow 3$.

Алгоритм Флойда-Уоршелла

Если надо только вычислить кратчайшие расстояния между всеми вершинами, но не требуется знать сами кратчайшие маршруты, можно использовать весьма элегантный и простой алгоритм Флойда-Уоршелла:

```
for (k = 0; k < n; k++)
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      if ( d[i][j] > d[i][k]+d[k][j] ) {
        d[i][j] = d[i][k]+d[k][j];
        p[i][j] = p[k][j];
      }
```

Сначала в матрице $\{d_{ij}\}$ записаны расстояния между вершинами напрямую. Затем рассмотрим все пути, которые проходят через первую вершину. Если путь через нее короче, чем напрямую, то заменяем значение в матрице на новый кратчайший путь. В конце элемент матрицы $\{d_{ij}\}$ содержит длину кратчайшего пути из *i* в *j*. Каждая строка матрицы $\{p_{ij}\}$ сначала заполняется так, как массив *c* в алгоритме Дейкстры, а в конце элемент $\{p_{ij}\}$ равен номеру предпоследней вершины для кратчайшего пути из вершины *i* в вершину *j*.

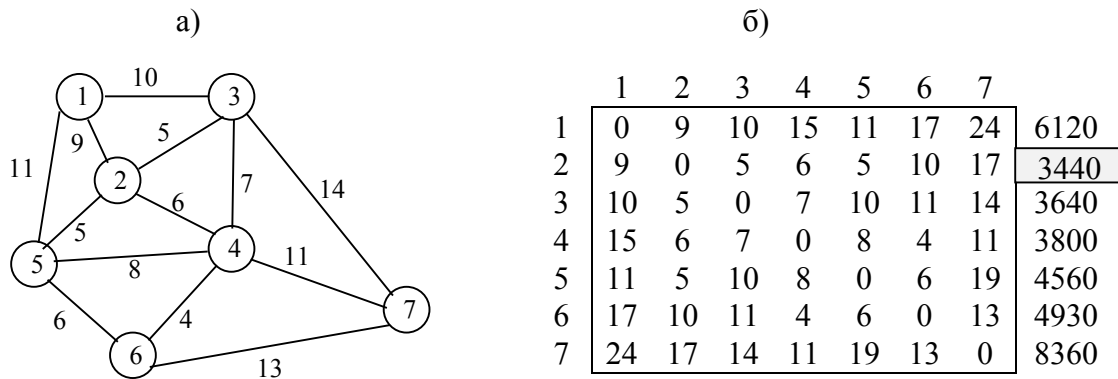
Оптимальное размещение

Задача на минимум суммы

Задача. Имеется *n* населенных пунктов, в каждом из которых живет p_i школьников ($i=1, \dots, n$). Надо разместить школу в одном из них так, чтобы общее расстояние, проходимое всеми учениками по дороге в школу, было минимальным.

Эта задача просто решается на основе алгоритма Флойда-Уоршелла. Сначала получаем матрицу минимальных путей из каждой вершины в каждую $\{d_{ij}\}$. Пусть школа размещается в вершине с номером k . Тогда общее расстояние, которое должны пройти ученики из пункта j по дороге в школу, равно произведению расстояния между вершинами i и j на количество учеников в вершине j , то есть $d_{ij} p_j$. Суммируя эти произведения для всех вершин, найдем общее расстояние, которое будут проходить ученики, если школа будет в пункте i . Естественно, надо выбрать такую вершину k , для которой это число будет наименьшим.

Рассмотрим сеть, показанную на рисунке а). Цифры у дуг показывают расстояние между населенными пунктами. Количество учеников в пунктах 1..7 задано числами (80, 100, 140, 90, 60, 50, 40).



На рисунке б) показана матрица длин кратчайших путей. В последнем столбце сосчитано общее расстояние, проходимое учениками, если школа будет в данной вершине. Поскольку надо выбрать наименьшее расстояние, то школу надо размещать в вершине 2.

Минимаксная задача размещения

Имеется n населенных пунктов, в одном из которых надо разместить спецподразделение, которое должно выезжать на вызовы при срабатывании сигнализации на одном из охраняемых объектов. Надо расположить его так, чтобы самый дальний охраняемый объект достигался за минимальное время.

Прежде всего, находится матрица кратчайших путей. Затем в каждой строке i выбирается максимальное число – это будет расстояние до самого отдаленного охраняемого объекта, если пункт полиции разместить в вершине i . Затем выбирается такой пункт j , для которого это число наименьшее.

Более сложной является минимаксная задача, в которой допускается размещать объект **на ребрах сети** (между пунктами). Для решения этой задачи можно использовать следующие идеи.

Рассмотрим ребро между вершинами i и j . Если расположить пункт полиции на нем, то расстояние от него до другой вершины k не может быть меньше, чем

$$\delta_k = \min(d_{ik}, d_{jk})$$

Вычислив эти величины для всех вершин k , определим максимальную из них. Это число δ_0 называется **нижней оценкой ребра (i, j)** , поскольку при любом выборе места для пункта полиции на этом ребре расстояние до самого дальнего объекта не будет меньше δ_0 . Мы уже знаем минимально возможное расстояние до самого дальнего пункта при размещении отряда в одном из городов. Очевидно, имеет смысл рассматривать только те ребра, для которых нижняя оценка не превышает это расстояние.

Для каждого ребра (i, j) с нижней оценкой, меньше уже полученной, сделать следующие операции:

1. найти самую дальнюю вершину k из всех тех вершин, в которые надо ехать через вершину i (то есть $d_{ik} > d_{jk}$)
2. найти самую дальнюю вершину l из всех тех вершин, в которые надо ехать через вершину j (то есть $d_{il} < d_{jl}$)
3. вычислить середину расстояния между вершинами k и l ; если она попадает на ребро (i, j) и расстояние до самых дальних вершин меньше уже найденного, запомнить это место

Ниже этот алгоритм записан на языке Си:

```

minDist = 32767;
for (i = 0; i < N-1; i ++ )
    for (j = i+1; j < N; j ++ ) {
        maxDi = maxDj = 0;
        for (k = 0; k < N; k ++ ) {
            if ( k == i || k == j ) continue;
            if ( D[j,k] < D[i,k] ) {
                if ( D[j,k] < maxDj ) maxDj = D[j,k];
            }
            else
                if ( D[j,k] < maxDj ) maxDj = D[j,k];
        }

        dist = (maxDi + D[i][j] + maxDj) / 2;

        if ( dist < minDist && dist > maxDi &&
            dist < maxDi+D[i][j] ) {
            minDist = dist;
            ...
        }
    }

```

очень большое число

перебрать все ребра (i, j)

перебрать все остальные вершины

запомнить новый вариант



Задача коммивояжера



Формулировка задачи

Эта знаменитая задача была сформулирована в 1934 г. и в области дискретной оптимизации стала испытательным полигоном, на котором проверяются новые методы. Проблема заключается в том, что эта задача относится к многочисленному классу задач, которые теоретически не решаются иначе, чем перебором.

Задача коммивояжера. Коммивояжер (бродячий торговец) должен выйти из первого города и, посетив по разу в неизвестном порядке города $2, 3, \dots, n$, вернуться обратно в первый город. В каком порядке надо обходить города, чтобы замкнутый путь (тур) коммивояжера был кратчайшим?



Метод грубой силы

Этот метод предусматривает просто перебор всех вариантов. Сначала нам надо получить некоторую допустимую последовательность обхода вершин. Поскольку первый город (1)

зафиксирован, всего таких перестановок будет $(n-1)!$. Получив очередную перестановку надо найти длину пути и, если она окажется меньше уже найденного кратчайшего пути, ее надо запомнить и перейти к следующей перестановке.

Поскольку нами уже написана процедура получения всех перестановок из целых чисел от **1** до **N** (см. пункт *Комбинации и перестановки* раздела *Рекурсия* в главе III), имеет смысл ее использовать. Удобно ввести глобальные переменные

```
int Pmin[N],      // лучшая перестановка
    P[N],         // текущая перестановка
    Lmin,         // минимальная длина
    L,            // текущая длина
    D[N][N];      // матрица расстояний
```

Основная программа будет инициализировать глобальные переменные, вызывать рекурсивную процедуру построения оптимального пути и распечатывать результат:

```
void main()
{
    Lmin = 32767;
    L = 0;
    P[0] = 1;
    Commi(1);
    for ( int i = 0; i < N; i ++ )
        printf("%d ", Pmin[i]);
}
```

очень большое число

начальная вершина 1

построить лучший тур из вершины 1

Ниже приведена рекурсивная процедура, решающая задачу коммивояжера, приводится ниже. Текущая длина пути вычисляется последовательно с добавлением каждого нового звена.

```
void Commi( int q )
{
    int i, temp;
    if ( q == N ) {
        if ( L < Lmin ) {
            Lmin = L;
            for ( i = 0; i < N; i ++ )
                Pmin[i] = P[i];
        }
        return;
    }
    for ( i = q; i < N; i ++ ) {
        temp = P[q]; P[q] = P[i]; P[i] = temp;
        L += D [P[q-1]] [P[q]];
        Commi ( q+1 );
        L -= D [P[q-1]] [P[q]];
        temp = P[q]; P[q] = P[i]; P[i] = temp;
    }
}
```

q - число уже поставленных вершин

перестановка получена

новый минимальный тур

перестановка $P[q]$ и $P[i]$

добавить новое ребро

вернуть $P[q]$ и $P[i]$ на место

Главная трудность такого переборного способа заключается в том, что для больших n (например, для $n > 20$) число комбинаций настолько велико, что перебор нереален.

Однако, часто можно найти такую стратегию перебора, при которой сразу отбрасывается целая группа вариантов, которые заведомо не лучше уже найденного минимального решения. Одним из таких алгоритмов является метод ветвей и границ.



Метод ветвей и границ

Идея этого подхода проста и естественна: если частичная длина пути стала больше длины самого лучшего из уже найденных туров, дальнейшую проверку делать бессмысленно. При этом второй цикл `for` в процедуре приобретает вид

```
for ( i = q; i < N; i ++ ) {
    temp = P[q]; P[q] = P[i]; P[i] = temp;
    L += D [P[q-1]] [P[q]];
    if ( L < Lmin ) Commi ( q+1 );
    L -= D [P[q-1]] [P[q]];
    temp = P[q]; P[q] = P[i]; P[i] = temp;
}
```

Все варианты туров можно изобразить в виде дерева с корнем в вершине 1. На первом уровне находятся все вершины, в которые можно идти из вершины 1 и т.д. Если мы получили $L > Lmin$, то сразу отсекается целая ветка такого дерева.



Алгоритм Литтла

Эффективность метода ветвей и границ очень сильно зависит от порядка рассмотрения вариантов. При удачном выборе стратегии перебора можно сразу же найти решение, близкое к оптимальному, которое позволит "отсекать" очень большие ветви, для которых нижняя оценка больше, чем стоимость уже найденного кратчайшего тура. Литтл с соавторами на основе метода ветвей и границ разработали удачный алгоритм, который позволяет во многих случаях быстро решить задачу коммивояжера.

Идея очень проста - разбить все варианты на классы и получить **оценки снизу** для каждого класса (оценкой снизу называется такое число, что стоимость любого варианта из данного класса заведомо не может быть ниже него).

Продemonстрируем на примере применение метода Литтла. Пусть есть 6 городов и матрица а) задает стоимость c_{ij} проезда из города i в город j . Прочерк означает, что из города i в город i ходить нельзя.

а)	1	2	3	4	5	6
1	-	6	4	8	7	14
2	6	-	7	11	7	10
3	4	7	-	4	3	10
4	8	11	4	-	5	11
5	7	7	3	5	-	7
6	14	10	10	11	7	-

б)	1	2	3	4	5	6
1	-	2	0	4	3	10
2	0	-	1	5	1	4
3	1	4	-	1	0	7
4	4	7	0	-	1	7
5	4	4	0	2	-	4
6	7	3	3	4	0	-

в)	1	2	3	4	5	6
1	-	0	0	3	3	6
2	0	-	1	4	1	0
3	1	2	-	0	0	3
4	4	5	0	-	1	3
5	4	2	0	1	-	0
6	7	1	3	3	0	-

Предположим, что добрый мэр города j постановил выплачивать всем, кто въедет в его город 5 долларов. При этом из каждого элемента j -ого столбца матрицы надо вычесть 5. Так как в **каждом** туре надо въехать в город j ровно **один** раз, то все туры подешевеют одинаково, и оптимальный тур останется оптимальным. Так же если мэр города i будет выплачивать 5 долларов каждому **выехавшему**, из каждого элемента i -ой строки надо вычесть 5. Но все туры подешевеют одинаково, и это снова не повлияет на результат.

Если из всех элементов любой строки или столбца вычесть одинаковое число, то минимальный тур остается минимальным

Теперь вычтем минимальный элемент из каждой строки и каждого столбца матрицы а) (эта операция называется **приведением по строкам**) и получим матрицу б) (надо вычесть числа 4, 6, 3, 4, 3 и 7 из строк 1-6 соответственно). Затем из каждого столбца вычтем минимальный в нем

элемент (числа 2, 1 и 4 из столбцов 2, 4 и 6, соответственно). После такого **приведения по столбцам** получим матрицу в). Если нам удастся найти минимальный тур в этой матрице, то он же будет минимальным и в исходной, только к его стоимости надо прибавить сумму всех чисел, которые мы вычитали из строк и столбцов, то есть 34. Поскольку в матрице в) нет отрицательных чисел, то стоимость минимального тура в ней больше или равна нулю, поэтому **оценка снизу** всех туров равна 34.

Теперь начинается основной шаг алгоритма. Выполним **оценку нулей** следующим образом. Рассмотрим нуль в клетке (1,2) матрицы в). Он означает, что цена перехода из 1 в 2 равна 0. А если мы не пойдем из 1 в 2, то все равно придется въехать в 2 (за цены, указанные в столбце 2 - дешевле всего за 1 из города 6). Также придется выехать из города 1 (за цену, указанную в первой строке - дешевле всего за 0) в город 3. Таким образом, если не ехать из 1 в 2 "по нулю", надо заплатить минимум 1. Это и есть "**оценка нуля**".

В матрице г) поставлены все оценки нулей, которые не равны нулю. Выберем максимальную из этих оценок (в примере - любой нуль с оценкой 1, например в клетке (1,2)).

г)

	1	2	3	4	5	6
1	-	0 ¹	0	3	3	6
2	0 ¹	-	1	4	1	0
3	1	2	-	0 ¹	0	3
4	4	5	0 ¹	-	1	3
5	4	2	0	1	-	0
6	7	1	3	3	0 ¹	-

д)

	1	3	4	5	6
2	-	1	4	1	0
3	1	-	0 ¹	0	3
4	4	0 ¹	-	1	3
5	4	0	1	-	0
6	7	3	3	0 ¹	-

е)

	1	3	4	5	6
2	-	1	4	1	0 ¹
3	0 ³	-	0 ¹	0	3
4	3	0 ¹	-	1	3
5	3	0	1	-	0
6	6	3	3	0 ¹	-

Пусть выбрано ребро (1,2). Разобьем все туры на 2 класса - включающие ребро (1,2) и не включающие его. Про второй класс можно сказать, что туры этого класса "стоят" 35 и более - это нижняя граница класса. Чтобы исследовать дальше первый класс, вычеркнем из матрицы г) первую строку и второй столбец - получим матрицу д). Чтобы не было цикла, надо поставить запрет в клетке (2,1). Эту матрицу можно привести на 1 по первому столбцу, таким образом, оценка этого класса - 35. Оценим нули в этой матрице (см. матрицу е)).

Снова выбираем клетку (3,1) с наибольшей оценкой нуля, равной 3. Оценка для класса "без (3,1)" равна 35+3=38. Вычеркиваем строку 3 и столбец 1 и ставим запрет на преждевременное замыкание (2,3) (матрица ж)).

ж)

	3	4	5	6
2	-	4	1	0
4	0	-	1	3
5	0	1	-	0
6	3	3	0	-

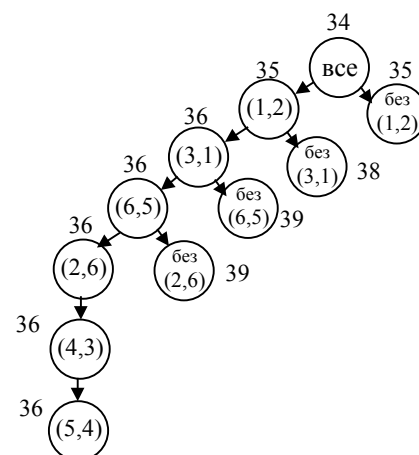
з)

	3	4	5	6
2	-	3	1	0 ¹
4	0 ¹	-	1	3
5	0	0 ²	-	0
6	3	2	0 ³	-

и)

	3	4	6
2	-	3	0 ³
4	0 ³	-	3
5	0	0 ³	-

Эта матрица приводится по столбцу 4 на 1. Таким образом, получаем матрицу з) и оценку класса 35+1=36. Нуль с максимальной оценкой 3 находится в клетке (6,5). отрицательный вариант имеет оценку 36+3=39. Для оценки положительно варианта вычеркиваем столбец 5 и строку 6, а также ставим запрет в клетку (5,6). Получим матрицу и), которая неприводима и не увеличивает оценку класса. Далее получаем ветвление по выбору ребра (2,6), отрицательный вариант имеет оценку 36+3=39. Далее вычеркиваем строку 2 и столбец 6 и ставим запрет в клетку (5,3) - получается матрица к). Теперь, когда осталась матрица 2 на 2 с запретами по диагонали, достраиваем тур ребрами (4,3) и (5,4). Мы получили тур 1-2-6-4-3-1



стоимостью в 36. На рисунке справа показано дерево ветвлений и оценки вариантов.

Теперь можно начать отбрасывать варианты. Все классы, имеющие оценку 36 и выше, лучшего тура не содержат. Поэтому отбрасываем все вершины дерева перебора, имеющие оценку 36 и выше, а также вершины, у которых "убиты" оба потомка. Осталось проверить, не содержит ли лучшего тура класс "без (1,2)". Для этого в приведенной матрице в) поставим запрет в клетку (1,2), приведем столбец 2 на 1 и оценим нули. Полученная матрица л) приведена ниже.

л)

	1	2	3	4	5	6
1	-	-	0 ³	3	3	6
2	0 ¹	-	1	4	1	0
3	1	1	-	0 ¹	0	3
4	4	4	0 ¹	-	1	3
5	4	1	0	1	-	0
6	7	0 ¹	3	3	0	-

м)

	1	2	4	5	6
2	0	-	4	0	1
3	-	1	0	0	3
4	4	4	-	1	3
5	4	1	1	-	0
6	7	0	3	0	-

Оценка нулей дает 3 для (1,3), поэтому стоимость отрицательного варианта $35+3=38$ превосходит стоимость уже полученного тура и отрицательный вариант отсекается. Для оценки положительного варианта вычеркиваем первую строку и 3-й столбец, ставим запрет в (3,1) и получаем матрицу м). Она приводится по строке 4 на 1, поэтому оценка положительного варианта равна $35+1=36$ и этот класс также отбрасывается. таким образом мы получили минимальный тур.

Хотя теоретических оценок быстродействия этого и подобных алгоритмов не получено, на практике они часто позволяют решить задачи с $n=100$.



Метод случайных перестановок

Для того, чтобы приближенно решать задачу коммивояжера для больших n , на практике часто используют метод случайных перестановок. В алгоритме повторяются следующие шаги:

1. Выбрать случайным образом номера вершин i и j в перестановке.
2. Если при перестановке вершин с номерами i и j длина пути уменьшилась, такая перестановка принимается.

Такой метод не гарантирует, что будет найдено точное решение, но очень часто позволяет найти приемлемое решение за короткое время в тех случаях, когда другие алгоритмы неприменимы.



Задача о паросочетаниях



Формулировка задачи

Задача. Есть m мужчин и n женщин. Каждый мужчина указывает несколько (от 0 до n) женщин, на которых он согласен жениться. Каждая женщина указывает несколько мужчин (от 0 до m), за которых она согласна выйти замуж. Требуется заключить наибольшее количество моногамных браков.

Для формулировки задачи в терминах теории графов надо ввести несколько определений.

Двудольным называется граф, все вершины которого разбиты на две доли (части), а ребра проходят только между вершинами разных частей.

Паросочетанием называется множество ребер, не имеющих общих вершин.

Таким образом, эта задача равносильна следующей:

Задача о наибольшем паросочетании. В заданном двудольном графе найти наибольшее паросочетание.

Для описания графа введем матрицу возможных браков $A=\{a_{ij}\}$, где $a_{ij}=1$, если возможен брак между мужчиной i ($i=1..m$) и женщиной j ($j=1..n$) (оба согласны) и $a_{ij}=0$, если брак невозможен.

Достижимость вершин в графе

Для начала решим вспомогательную задачу, которая понадобится в дальнейшем.

Задача о достижимости. Найти, какие вершины достижимы в заданном ориентированном графе из данной вершины s .

Эта задача решается с помощью алгоритма, очень похожего на алгоритм Дейкстры. Заведем три вспомогательных массива Q , R и P размером n , где n - число вершин:

- в массив Q будем записывать номера вершин в порядке их просмотра (начиная с s)
- в массиве R элемент $R_i=1$, если уже найден путь из s в i , и $R_i=0$, если ни один путь не найден
- в массиве P элемент P_i есть номер предпоследней вершины на пути от s к i , или $P_i=-1$, если ни один путь не найден

Две переменные a и z обозначают начало и конец "рабочей области" массива Q . Теперь можно привести алгоритм. Он состоит из двух частей.

Инициализация

Начать с вершины s , ни одна из вершин не рассмотрена:

1. Присвоить $a=0$; $z=0$; $Q[0]=s$;
2. Для всех i присвоить $R[i]=0$, $P[i]=-1$;

Общий шаг

В цикле рассмотреть все вершины орграфа, которые достижимы непосредственно из $Q[a]$. Если путь до вершины k еще не был найден (то есть $R_k=0$), сделать

1. $z++$; $Q[z] = k$; $P[k] = Q[a]$; $R[k] = 1$;
2. $a++$;

Повторять общий шаг пока $a \leq z$.

Покажем работу алгоритма на примере. Найти вершины данного графа, достижимые из вершины 1. Последовательно имеем

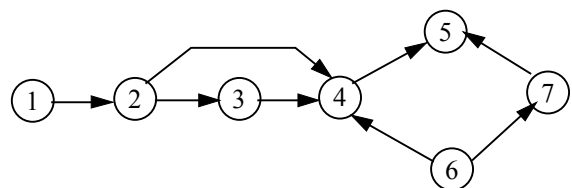
Рассмотрена вершина 2:

- $Q = \{1, 2\}$
- $R = [0, 1, 0, 0, 0, 0, 0]$
- $P = [-1, 1, -1, -1, -1, -1, -1]$

Рассмотрены вершины 3 и 4:

- $Q = 1, 2, \{3, 4\}$
- $R = [0, 1, 1, 1, 0, 0, 0]$
- $P = [-1, 1, 2, 2, -1, -1, -1]$

Рассмотрена вершины 5:



- $Q = 1, 2, 3, 4, \{5\}$
- $R = [0, 1, 1, 1, 1, 0, 0]$
- $P = [-1, 1, 2, 2, 4, -1, -1]$

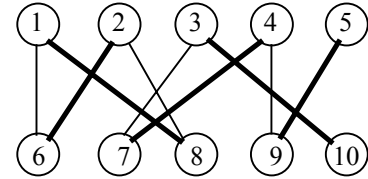
После этого новых достижимых вершин нет, рабочая зона массива Q (в фигурных скобках) закончилась.

Массив P можно раскрутить как в алгоритме Дейкстры. Например, вершина 5 достижима из 4, вершина 4 - из 4, а 2 - из 1, поэтому существует путь 1-2-4-5.



Метод чередующихся цепей

Задача о наибольшем паросочетании решается методом чередующихся цепей. Пусть M - некоторое паросочетание в двудольном графе B . Ребра, которые входят в M , будем обозначать толстыми линиями, а все остальные - тонкими. Цепь, в которую входят поочередно жирные и тонкие ребра, будем называть чередующейся. Например, цепь (1, 6, 2, 8) - чередующаяся. По определению цепь из одного ребра тоже чередующаяся.



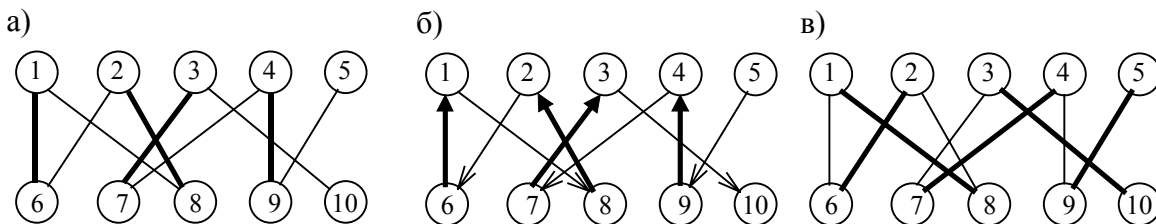
Вершины, которые связаны с жирными ребрами (участвующими в паросочетании), называются **занятыми** или **насыщенными**. Остальные вершины называются **свободными** или **ненасыщенными**.

Очевидно, что если в графе есть цепь, которая начинается и заканчивается на свободной вершине, то в ней тонких ребер на 1 больше, чем жирных. Поэтому такую цепь можно "перекрасить", то есть заменить все жирные ребра тонкими и наоборот. При этом пар станет на 1 больше, то есть решение улучшится. Такая цепь называется **увеличивающейся**.

Можно доказать, что паросочетание является наибольшим тогда и только тогда, когда в графе нет увеличивающихся цепей. На этой идее основан следующий алгоритм.

1. Построим начальное паросочетание "жадным" алгоритмом: будем брать по очереди незанятые вершины первой части и, если существует допустимое ребро, вводить его, не думая о последствиях.
2. Строим ориентированный граф так: все дуги, вошедшие в паросочетание ("жирные") направляем вверх, а все остальные - вниз.
3. Просматриваем все свободные вершины первой части. Если среди достижимых вершин есть незанятая вершина из второй части, то есть увеличивающаяся цепь. Ее надо увеличить и перейти снова к пункту 2.
4. Если увеличивающихся цепей нет, получено наибольшее паросочетание.

Например, граф на рисунке а) (паросочетание получено "жадным" алгоритмом) после введения ориентации принимает вид б).



Из свободной вершины 5 достижимы вершины 7, 3, 4, 9 и 10, причем из них вершина 10 свободна, то есть существует увеличивающаяся цепь (5, 9, 4, 7, 3, 10). После ее увеличения получаем наибольшее паросочетание (рисунок в).