Введение в C и C++.
Алгоритмы и структуры данных в примерах.

10 марта 2004 г.

Введение

Эта книга посвящена знакомству с языками программирования C и C++, базовыми алгоритмами и структурами данных. Следует отметить, что цель книги не научить языку программирования, а познакомить с основными конструкциями и стилем программ на C и C++, дать материал для анализа, подражания и критики.

Книга будет полезна не только тому, кто только начал изучать программирование, но и искушенному в программировании читателю — ему будет интересна вторая часть книжки, где речь идет об алгоритмах и структурах данных. Там, кроме задачи сортировки, рассматриваются различные практические, но по-прежнему фундаментальные задачи, которые решаются с помощью использования хэш-таблиц, деревьев поиска, сбалансированных деревьев, бинарной кучи, и других структур. Также там рассказано про важнейшие идеи решения алгоритмических задач — это рекурсия, "разделяй и властвуй", динамическое программирование и "жадные" алгоритмы.

Учить язык программирования — это особенная задача, отличная от изучения географии, математики или физики. Есть целый ряд аргументов, что лучший способ учить новый язык (как иностранный, так и язык программирования) — это ассоциативный метод, основанный на живых примерах. То есть не логическое последовательное изучение основ и подробностей синтаксиса, анализа структуры и классификация объектов, а некоторое непоследовательное, проблемно-ориентированное знакомство с языком, от простого к сложному, с постепенным погружением в детали языка, которые в том или ином случае оказываются нужны.

Конечно, метод погружения имеет свои недостатки. И в конечном итоге, как начинающему, так и профессиональному программисту всегда нужно иметь под рукой полноценный, последовательный справочник и документ со спецификацией языка.

Цель данной книги — дать несколько наглядных примеров, которые бы познакомили читателя с языком C и основными его конструкциями, а также показать реализации классических структур данных и алгоритмов для решения популярных задач.

Оглавление

Глава 1

Что такое языки C и C++? Зачем они нужны?

Прежде, чем перейти к рассказу о языках C и C++, хочется предупредить читателя. Языки C и C++ — чрезвычайное мощное, гибкое средство, с помощью которого можно решать алгоритмические и вычислительные задачи любой сложности, осуществлять моделирование физических систем, создавать распределенные компьютерные системы с практически любой функциональностью, разрабатывать приложения, использующие различные сетевые протоколы и взаимодействующие с базами данных, с красивым и удобным графическим интерфейсом.

Языки C и C++ универсальны. В этом их плюс и в этом же их минус.

Надо понимать, что язык, специально созданный для моделирования физических систем может оказаться удобней и адекватней при решении задачи моделирования, нежели язык C. А языки и технологии, придуманные для разработки WEB-систем, удобнее при разработке WEB-системы. И так далее, языки, созданые специально для решения определенной задачи очевидно будут лучше в своей области, нежели какие бы то ни были универсальные средства.

В начале XXI века программисты в своем инструментарии имели уже около 1000 различных языков программирования, и каждый из этих языков был достоин внимания при решении задач определенного класса. Языки программирования как различные виды животных в одной экосистеме — некоторые из них похожи, други совсем не похожи, ниши некоторых видов пересекаются, ниши других – совсем не связаны друг с другом. Очень часто один язык вытесняет другой, также наблюдается постоянная эволюция языков. Языки могут отличатся как классом эффективно решаемых ими задач, так и парадигмой, на базе которой построены.

Язык C — это не объектно-ориентированный, императивный, типизированный, низкоуровневый язык программирования. Язык C++ является его обектно-ориентированным расширением.

То, что язык C ***низкоуровневый***, означает несколько вещей:

- Команды и типы данных, которые заложены в C изначально, элементарны (на основе них можно создавать свои, сколь угодно сложные функции и структуры данных).
- Основная информационная единица, с которой работает программист на C – это байт. Это значит, что он знает о том, что представляют данные в его программе на уровне байтов и может “добратся” до любого их байта.
- Программист на C представляет, как его программа будет расположена в памяти.
- Программист на C может перейти на самый низкий уровень программирования — делать в свой программе “ассемблерные вставки” и писать на уровне команд процессора.
- Язык C приспособлен для того, чтобы работать непосредственно с любыми устройствами компьютера.

Прилагательное ***императивный*** означает, что в языке C логику работы программы представляют в виде глаголов (действий) : “пока выполнено условие ... делать действия ...”, “выполнить последовательность действий ... N раз”, “если выполнено условие ... перейти к действию ..”

В отличие от C язык Perl высокоуровневый, нетипизированный и скриптовой:

- В Perl изначально заложено несколько высокоуровневых команд и структур, например команда сортировки `sort` или преобразования элементов массива `map`.
- Элементарный тип данных в Perl — это строчка символов (в то время как в языке C строчка — это массив символов).
- В языке

Например, на Perl можно написать:

```
print map {“$_ = $ENV{$_}\n” }  (sort keys %ENV);
```

Эта программа, состоящая из одной строчки, печатает названия и значения переменных окружения системы (Environment viriables), упорядочив их в алфавитном порядке.

В языке C нет команды сортировки и нет встроенных Хэш-таблиц, но, как и в большинстве языков, можно создавать свои структуры данных и библиотеки функций. Так в стандартной библиотеке `stdlib` есть функция `qsort`, которой можно пользоваться — не нужно писать алгоритм сортировки самому. Не несмотря на это, код программы на C, печатающей в оказывается существенно объемнее и труднее для понимания новичка:

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

extern char **environ;

int
cmp(const void *a, const void *b)
{
    return strcmp(*(char**)a, *(char**)b);
}

int
main() {
    int i = 0;
    while(environ[i] != NULL) i++;
    qsort(environ, i, sizeof(char*), cmp);

    i = 0;
    while(environ[i] != NULL) {
        printf(“%s\n”, environ[i++]);
    }
    scanf(“%d”, &i);
    return 0;
}
```

Следует упомянуть, это программа печатающая переменные окружания уже написана за нас и встроенна в shell как Linux, так и Windows — команда `set`.

Вообще, такие алгоритмы как сортировка и вычисление математических функций, а также популярные структуры данных давно уже написаны — осталось только научится ими правильно пользоваться. Поэтому с уровня глаголов – “сложить два числа”, “обнулить значение переменной”, “поставить белую точку на черном экране” – мы можем перейти на более высокий уровень: “создать окошко”, “найти корень функции”, “нарисовать график функции”, “получить таблицу из базы данных” и т.д.

Эта книга посвящена программированию на низком уровне, то есть тому, чем вам, возможно, никогда не приедется заниматься — для решения своих задач вы будете пользоваться высокоуровневым и, скорее всего, объектно ориентированным языком программирования.

Но с другой стороны, программировать на высоком уровне и не уметь программировать на низком уровне, это все равно что оперировать с эллиптическими интегралами и при этом не уметь складывать и умножать числа.

Надо понимать, что польза от изучения языка программирования низкого уровня заключается не в обретении инструмента для зарабатывания денег, а в прояснении мышления. Опыт программирования на C развивает ***алгоритмическое мышление***, которое стоит в ряду с абстрактным, логическим, аналитическим и геометрическим типами мышления, и даже, в определенном смысле, давлеет над ними, находясь на одну ступеньку выше.

Опыт программирования дарит вашему разуму новые идеи и метафоры, обогащает его методами поиска решения, и развивает способность к рефлекии и абстракции.

ПРАКТИКА ПРОГРАММИРОВАНИЯ ДЕЛАЕТ МЫШЛЕНИЕ ЯСНЫМ

Это цель, ради которой стоит изучать язык C.

Ворожцов Артем

Глава 2

Знакомство с языком C

Компиляция

Программа на C это один или несколько текстовых файлов. То есть программы можно писать с помощью любого текстового редактора.

Пока, мы с вами будем рассматривать программы состоящие из одного файла.

После того, как программа написана, нужно создать запускаемый файл. Если ваша программа есть файл `hello.c`, то для компиляции его компилятором GNU нужно выполнить команду:

```
gcc hello.c -o hello
```

В результате получится файл `hello`, который пожно запускать (по-английский — **exe**cute).

Здравствуй, мир!

Первая программа, которую мы рассмотрим, — это "Hello, world" — программа, которая напечатает на экран строчку "Hello world!" и закончит своё выполнение:

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

Посмотрим на неё внимательно. Первая строчка `#include<stdio.h>` означает "подключи библиотеку, где определяются функции связанные с выводом и считыванием данных"[1]1. Аббревиатура STDIO означает "STanDard functions for Input and Output". Буква "h" после точки означает "header", то есть *заголовочный файл*. В заголовочных файлах описано какие функции предоставляет соответствующая библиотека. В действительности, `#include<...>` есть ***директива препроцессора***, то есть команда, которая выполняется до начала компиляции файла. Смысл этой директивы заключается в том, чтобы вставить в файл программы содержимое другого файла, имя которого указано в угловых скобках. Обычно заголовочные файлы содержат только ***прототипы фукций***, то есть просто список функций с указанием аргументов и типа возвращаемого значения.

Далее идет функция `main`. Она начинается с объявления

```
int main(void)
```

[1] 1

что соответствует

“функция с именем `main`, которая
возвращает целое число (число типа **int**) и у которой нет аргументов (**void**)”

Слово **void** можно переводить как **ничто**. Далее открываются фигурные скобки и идет описание этой функции, в конце фигурные скобки закрываются. Функция `main` — эта главная функция вашей программы, именно она начинает выполняться, когда ваша программа запускается.

Между фигурных скобок находится *тело* функции — в нем описана логика — то что эта функция делает. Наша функция делает одно единственное действие —

```
printf (“Hello, world!\n”);
```

Это вызов функции `printf` из библиотеки `stdio`. В результате выполнения функция напечатает на экран текст `Hello, world!`. Обратите внимание на комбинацию символов `\n` — она задает специальный символ, символ, который в действительности не символ, а действие — закончить строчку и перейти на следующую строчку. Таких специальных символов несколько, все они начинаются на \ (символ backslash).

Затем идет команда **return** `0;` которая завершает выполение функции и возвращает значение 0. Функция `main` должна возвращать 0, если выполнение прошло успешно.

Учимся складывать

Разнообразные вычисления — моделирование, решение алгебраических и дифференциальных уравнений — это то, для чего, собственно, и создавались первые компьютеры. Давайте и мы научимся использовать компьютер для вычислений. Начнём со сложения двух чисел.

В нашей программе будет две целочисленные переменные и b — две ячейки памяти, в которых могут хранится целые числа из определенного диапазона.

Переменные объявляются сразу же после открывающей фигурной скобочкой функции `main`. Объявления начинаются со слова, указывающего тип переменных.

В языке C есть несколько типов данных. Они делятся на две группы: целые типы и типы с плавающей точкой. К первому типу относятся **char**, **short**, **int**, **long**, а также **unsigned char**, **unsigned int**, **unsigned long**. Ко второму — **float**, **double** и **long double**.

Тип	Обозначение	Длина	Область
char	`%c`	8 бит	-128 .. 127
int	`%d`	16 бит	-32768 .. 32767
long	`%ld`	32 бит	$-2\,147\,483\,648$ .. $2\,147\,483\,647$
float	`%f`	64 бит	$\pm(1.7 \cdot 10^{-308}$ ... $1.7 \cdot 10^{308})$
double	`%lf`	64 бит	$\pm(1.7 \cdot 10^{-308}$ ... $1.7 \cdot 10^{308})$
long double	`%Lf`	80 бит	$\pm(3.4 \cdot 10^{-4932}$ ... $3.4 \cdot 10^{4932})$

Таблица 2.1: Типы данных.

```
#include <stdio.h>
int main ()
{
    int a, b;
```

```
    printf (“Введите два числа: ”);
    scanf (“%d%d”, &a, &b);
    printf (“%d”, a + b)
    return 0;
}
```

Функция `scanf`, также как и `printf`, функция из библиотеки `stdio`. Эта функция считывает данные, которые пользователь (тот, кто запустит вашу программу) вводит с клавиатуры. Слово *scan* означает *считывать данные*, а *print* — печатать данные. Буква “f” на конце соответствуют английскому слову *formated*, то есть `scanf` и `printf` есть функции для форматированного ввода и вывода данных.

Первый аргумент у функции `scanf` (то, что идет после открывающей круглой скобки до запятой) — это описание формата входных данных, то есть описание того, что мы собственно ожидаем от пользователя. В этой программе мы ожидаем, что пользователь введет два целых числа. Символ `%` служебный, с него начинается описание формата. Обычно, после него идет один или два символа, определяющих тип входных данных.

В таблице 2.1 приведены некоторые типы данных. Для каждого типа с указаны промежуток значений, количество бит, которые занимает одна переменная этого типа, и наиболее популярное обозначение формата данных для ввода и вывода значения переменной этого типа. Следует отметить, что для каждого типа данных, существует несколько форматов, и наоборот, для разных типов можно использовать один и тот же формат, хотя это редкость.

Приведенная программа умеет складывать только целые числа. Если вы хотите складывать действительные числа, то эту программу нужно несколько модифицировать. Ниже приведена программа, которая считывает два действительных числа и выводит результат четырех арифметических операций: сложения, вычитания, умножения и деления. Причем, программа выводит результаты вычислений два раза — сначала в обычном виде, а потом со специальным форматированием. Формат **“%10.3lf”** соответствует выводу числа типа **double**, при котором под запись числа числа выделяется ровно 10 позиций (если это возможно), а после запятой пишется ровно три знака. Равнение происходит по правому краю.

```
#include <stdio.h>
int main ()
{
    double a, b;
    printf (“Введите два числа: ”);
    while(scanf (“%lf%lf”, &a, &b) == 2 )
    {
        printf (“%lf %lf %lf %lf\n”, a + b, a - b, a * b, a / b );
        printf (“%10.3lf %10.3lf %10.3lf %10.3lf\n”, a + b, a - b, a * b, a / b );
    }
    return 0;
}
```

В этой программе мы встречаемся с оператором **while**. Конструкция

while `( A )` `B;`

означает буквально следующее

Пока выполнено условие `A` делать `B`.

В нашем случае

A = **scanf**(“%lf%lf”, &a, &b) == 2.

Что переводится как

“пользователь ввел два действительных числа и оди удачно считаны в переменные a и b”

Поэтому эта программа будет считывать пары чисел и выводить результаты арифметических операций, пока пользователь не введет что-нибудь непохожее на число. Цикл **while** закончится, когда функция scanf не сможет успешно считать два чисел.

Заметьте, что после каждой команды стоит точка с запятой. Одна из самых популярных синтаксических ошибок начинающих программистов — это не ставить точку запятой в конце команды.

Максимум

Программа “max.c” сначала узнаёт сколько у вас есть чисел, и заносит это в переменную n. Потом она считывает n чисел и в конце выдает максимальное из введённых чисел. Заметьте, что числа вы можете вводить, разделяя их пробелом <SPACE>, символом табуляции <TAB>, или нажимая после каждого числа <ENTER>. Символы <SPACE>, <TAB>, <ENTER> называются пробельными символавми. Функция scanf считывает объекты разделенные любым числом пробельных символов (white space).

```
#include <stdio.h>
int main ()
{
    int i, n, a, max;
    printf (“Введите количество чисел: ”);
    scanf (“%d”, &n);
    printf (“Введите %d чисел: ”, n);
    scanf (“%d”, &max);
    for(i = 1; i < n ; i++)
    {
        scanf (“%d”, &a);
        if(a > max) max = a;
    }
    printf (“%d”, max);
    return 0;
}
```

Обратите внимание на второй **printf** — он имеет два аргумента. Первый аргумент — это строчка (текст в двойных кавычках), которая задает, что будет печататься. В этой строчке встречается выражение %d, которое соответствует выводу десятичной записи целого числа. На месте этого выражения будет напечатано на экране компьютера значение второго аргумента, которое будет интерпретироваться как целое число.

В программе max.c мы впервые встречаемся с *условным оператором* **if** и *оператором цикла* **for**. Оператор условного перехода записывается так:

if(A) B;

Он соответствует предложению “Если выполнено условие **A**, то сделать **B**”. Оператор **for** устроен следующим образом:

for(A ; B ; C) D;

Элемент `D` может быть как одной командой, так и произвольным набором команд, заключенных в ***блок***. Команды объединяются в блок с помощью заключения их в фигурные скобок. В нашем случае `D` это

```
{
    scanf (“%d”, &a);
    if(a > max) max = a;
}
```

Элемент `D` называется телом цикла — это то, что будет выполнятся несколько раз. Сколько именно? Это зависит от `B` — тело `D` будет выполнятся пока выполнено условие `B`.

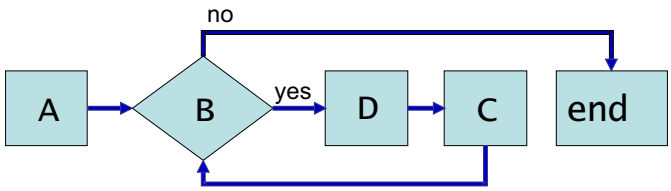
Например, строчка

```
for( i = 0 ; i < 10 ; i++ ) { printf ( “Hi!” ); }
```

означает “10 раз напечатать слово `Hi!`”. Более подробно:

1. положим `i = 0`;
2. проверим, что `i < 10`; если условие не выполнено, то перейдем к пункту 5;
3. выполним `printf ( “Hi!” )`;
4. выполним `i++` (увеличить переменную `i` на 1) и перейдем к пункту 2;
5. конец цикла;

Логику оператора **for** можно изобразить в виде блок схемы:



`for( A ; B ; C ) D;`

Логика нашей программы следующая:

1. Объявляем все переменные, которые встретятся встречаются в нашей программе:
   `n` : количество чисел,
   `i` : переменная цикла,
   `max` : переменная для хранения текущего найденного максимума,
   `a` : переменная с очередным числом, в неё мы считываем каждое число.
2. Печатаем на экран приглашение ввести число n.
3. Считываем n.
4. Печатаем на экран приглашение “Введите n чисел” (заметьте, что на экране будет не символ `n`, а само число n).
5. Считываем первое из этих n чисел в переменную `max`.
6. В цикле $n - 1$ раз считываем очередное число, и если оно больше, чем текущий максимум, то обновляем переменную `max`.

7. Выводим значение переменной `max`.

Вы уже наверное заметили, что когда мы в **scanf** указываем переменные, в которые нужно поместить считываемые данные, то перед именем переменной ставим знак `&` (амперсанд), в то время как в **printf**, при печати значений переменных амперсанд не ставится. Операция "амперсанд" — это операция взятия адреса. Если **a** есть значение переменной **a**, то **&a** — это адрес в памяти компьютера, где хранится переменная **a**: одно есть само число, другое — место, где это число хранится. Функции **scanf** нужно знать не текущее значение переменной, а её адрес, чтобы положить туда результат считывания. А функции **printf** нужно знать само значение переменной, чтобы его напечатать. Запомните: при считывании чисел, функции **scanf** обязательно нужно передавать адрес переменной.

Таблица умножения

Следующая программа выводит на экран таблицу умножения. Переменные `i` и `j` соответствуют номеру строчки и номеру столбца. Здесь мы встречаемся с циклом в цикле. Переменная внешнего цикла j сначала равна 1. Начинает работу внутренний цикл. Переменная внутреннего цикла j меняется от 1 до n включительно. И при этом, для каждого значения j печатает на экран результат произведения `i * j`. формат вывода определяется как **"%5d"**, что означает печатать целое число, выделяя под него 5 позиций. При этом, если в числе меньше, чем пять разрядов, то оно придвигается к правому краю, а слева добавляются пробелы. Когда заканчивается внутренний цикл, происходит переход на новую строчку (команда `printf ("\n")`); После этого переменная внешнего цикла `i` увеличивает своё значение на 1 и становится равна 2. Затем снова запускается внутренний цикл, и печатается следующая строчка таблицы умножения и так далее. Пример работы программы (число 5 ввел пользователь программы):

```
> ./table
Введите n: 5
    1    2    3    4    5
    2    4    6    8   10
    3    6    9   12   15
    4    8   12   16   20
    5   10   15   20   25
>
```

```
#include <stdio.h>
int main()
{
    int i, j, n;
    printf ("Введите n: ");
    scanf ("%d", &n);

    for(i = 1; i <= n ; i++)
    {
        for(j = 1; j <= n ; j++)
        {
            printf ("%5d", i * j);
        }
        printf("\n");
```

```
    }
    return 0;
}
```

Вопросы и задания

1. Строчку

```
for(j = 1 ; j <= n; j++)
```

замените на

```
for(j = n ; j > -n; j--)
```

и посмотрите что получится.

2. Строчку

```
printf("%5d", i * j)
```

замените на

```
printf("%05d ", i * j)
```

и посмотрите, что получится.

ASCII

В языке C есть тип **char** для символов. Каждому символу сопоставлено число от 0 до 255, которое называется ASCII-кодом символа. Например, символу 'A' соответствует число 65. Символами можно оперировать, как числами и, наоборот, переменные типа **int** можно интепретировать как символы (сравнивать с символами на равенство или печатать как символы).

Для того, чтобы считывать один символ в библиотеке `stdio` есть функция `getchar`. Алгоритм программы следующий. Считывается символ и печатается в двух форматах: как символ (формат **"%c"**) и как число (формат **"%d"**). Это делается до тех пор, пока символ (а точнее его ASCII код) не будет равен 27, то есть пока не будет нажата клавиша <ESC>.

Выражение `ch != 27` означает логическое $\texttt{ch} \neq 27$.

```
#include <stdio.h>
void main ()
{
    int ch;
    do {
        ch = getchar();
        printf ("Вы нажали %c. ASCII код = %d\n", ch, ch);
    } while (ch != 27);
}
```

Вопросы и задания

1. Напишите программу, которая печатает все символы и их ASCII-коды.
2. Попробуйте напечатать как символ число больше 255. Что получается?

Перевод в верхний регистр

В следующей пробрамме введенная пользователем строчка переводится в верхний регистр, то есть те символы которые были в вернем регистре (заглавные буквы) остаются в верхнем

регистре, а те которые были в нижнем регистре, становятся в верхнем. Символы, не являющиеся латинскими буквами не меняются.

Надо сказать, что функцию перевода строчки в верхний регистр, конечно же, писать самому не надо, поскольку это стандартная функция, которая есть во многих библиотеках различных языков, а в большинство скриптовых языках она просто встроена. В частности, в стандартной библиотеке `string` реализованы функции `strlwr` и `strupr`, которые переводят строчки в нижний и в верхний регистр. Соответствующие функции для преобразования одного символа `tolower` и `toupper` определены в библиотеке `ctype`.

```
#include <stdio.h>
#define N 100
int main()
{
    char a[N];
    int i;
    scanf (“%s”, a);
    for(i = 0; a[i] != 0; i++)
    {
        if( a[i] <= ’z’ && a[i] >= ’a’)
        {
            a[i] += ’A’ - ’a’;
        }
    }
    printf (“%s”, a);
    return 0;
}
```

А вот программа, которая использует функцию `tolower` библиотеки `ctype` и переводит строчку в нижний регистр.

```
#include <string.h>
#include <stdio.h>
#include <ctype.h>
int main()
{
   int length, i;
   char[] string = “ThIs Is A sTrInG”;
   length = strlen(string);
   for (i = 0; i < length; i++)
   {
       string[i] = tolower (string[i]);
   }
   printf (“%s\n”, string);
   return 0;
}
```

Скобки

Программа “Скобки” определяет правильность введённой скобочной структуры. ***Вход.*** Слово в алфавите из двух круглых скобочек (и). Длина слова меньше 100000 символов. ***Выход.*** Либо `NO`, либо `YES`. Примеры работы программы:

ВХОД	ВЫХОД
()	YES
)(	NO
()(())()	YES
((((((()))))))	YES
())(()	NO

```
#include <stdio.h>
void main ()
{
    int ch;
    printf ("Введите слово из скобок: ");
    do {
        ch = getchar();
        if( ch == '(' ) a++;
        else if( ch == ')' ) if(--a < 0) break;
    } while(ch != '\n');
    if( a == 0) printf ("Правильно\n");
    else printf ("Не правильно\n");
}
```

Сдесь нам встречается оператор `==`, который соответствует логическому "тождественно равно". Когда мы пишем `i = 0`, то мы послулируем равенство, то есть присваеваем переменной `i` значение 0. Выражение `i == 0` является проверкой равенства, её значение равно "истина" или "ложь". В языке C/C++ "истина" соответствует любому ненулевому числу. Поэтому, вместо `i == 0` мы могли бы написать равносильное `i - 0`, но это не рекомендуется делать.

Обратите внимание на то, что программа не хранит скобочную структуру в памяти. В этом нет необходимости, поскольку алгоритм ***однопроходный***, то есть достаточно конечной памяти и одного пробега по сколь угодно большим входным данным, чтобы получит результат. Свойством однопроходности обладает также рассмотренный нами алгоритм поиска максимума из чисел. Задача сортировка чисел не является однопроходным алгоритмом.

Вопросы и задания

1. Какие из перечисленных задач имеют однопроходные решения:
   1) Найдите среднее арифметическое чисел.
   2) Найдите дисперсию чисел (средний квадрат отклонения от среднего).
   3) Найдите три самых маленьких числа среди введенных.
2. Напишите программу, которая проверяет правильность скобочной структуры, составленной из нескольких типов скобок (круглых, квадратных и фигурных). Например, `({()[]})` — правильная структура, а `({()[)}]` — не правильная.

Факториал

Факториал числа n это произведение первх n натуральных чисел:

$$n! = 1 \cdot 2 \cdot \ldots \cdot n.$$

В приведенной ниже программе определена функция факториал. Это определение основано на следующей ***рекуррентной*** (или ***рекурсивной***) формуле:

$$n! = n \cdot (n-1)!, \quad 0! = 1.$$

```
#include <stdio.h>
long factorial(long x)
{
    if( x == 0 ) return 1;
    return x * factorial (x - 1);
}
void main()
{
    long n;
    while( scanf("%ld", &n) == 1)
        printf("%ld\n", factorial (n));
}
```

Если бы не было строчки

```
if( n == 0 ) return 1;
```

то функция `factorial` постоянно бы вызывала саму себя, и в конце концов, мы получили бы сообщение об ошибке типа "error: переполнение стэка" или "error: превышена глубина рекурсии". Для того, чтобы этого не было, необходимо чтобы при некоторых значения аргумента, функция не вызывала бы саму себя. Но это, не является достаточным условием.

Идея рекурсии заключается в сведении задачи к этой же задаче, но для более простых аргументов (например для меньшего значения аргумента n). Со процесс сведения задачи к предыдущей должен когда нибудь заканчиваться. Поэтому рекурсивные функции для простейших входных данных должны знать, чему они равны и не пытаться их свести к чему-нибудь более простому.

Вопросы и задания

1. Улучшите программу, добавив к ней защиту от дурака — если $n < 0$, то программа должна по-прежнему адеватно работать, например, сообщать, что факториал для отрицательных чисел не определен, и не вызывать Runtime error — ошибку выполнения программы.

Степень числа

Для вычисления функции a^n тоже есть рекуррентная формула:

$$a^n = a \cdot a^{n-1}, \quad a^0 = 1.$$

А вот программа, которая основана на этой формуле:

```
#include<stdio.h>
double power(double x, long n)
{
    if(n == 0) return 1;
    if(n < 0) return power ( 1 / x, -n);
    return x * power(x, n - 1);
}
void main()
{
```

```
    double x;
    long n;
    while (scanf ("%lf %ld", &x, &n) == 2)
        printf("%lf\n", power (x, n));
}
```

Но у функции степени есть более "умная" рекурсивная функция:

$$a^n = \begin{cases} a \cdot a^{n-1}, & \text{если } n \text{ нечетная,} \\ (a^{n/2})^2, & \text{если } n \text{ четная.} \end{cases}$$

Например, если обозначть стрелочкой $\to$ слово "сводится к ", то при вычислении a^{12} для первой рекурсии получим цепочку длины 12:

$$a^{12} \to a^{11} \to a^{10} \to a^9 \to a^8 \to a^7 \to a^6 \to a^5 \to a^4 \to a^3 \to a^2 \to a^1 \to a^0.$$

А для второй рекурсии цепочку из 5 шагов:

$$a^{12} \to a^6 \to a^3 \to a^2 \to a^1 \to a^0.$$

Для больших n разница в длине цепочки более разительная. В частности a^{10000} первой рекурсией вычисляется за 10000 шагов, а второй за 19 шагов.

```
double power(double x, long n)
{
    double tmp;
    if(n == 0) return 1;
    if(n < 0) return power ( 1 / x, -n);
    if(n % 2) return x * power (x, n - 1);
    return power(x * x, n / 2);
}
```

Вопросы и задания

1. Сколько шагов требуется для вычисления a^{30} вторым методом?
2. Покажите, что второй алгоритм выполняется за логарифмическое по n число шагов, а точнее ограничено сверху $2 \cdot \log_2 n$, а еще точнее, в точности равно числу знаков в двоичной записи числа n плюс число единичек в этой записи.

Треугольник Паскаля

Всем известны формулы

$(a+b)^0$	$=$	1	0						1					
$(a+b)^1$	$=$	$a+b$	1					1		1				
$(a+b)^2$	$=$	$a^2+2ab+b^2$	2				1		2		1			
$(a+b)^3$	$=$	$a^3+3a^2b+3ab^2+b^3$	3			1		3		3		1		
$(a+b)^4$	$=$	$a^4+4a^3b+6a^2b^2+4ab^3+b^4$	4		1		4		6		4		1	
$(a+b)^5$	$=$	$\dots$	5	1		5		10		10		5		1
$(a+b)^6$	$=$	$\dots$	6	1	6		15		20		15		6	1

То, что нарисовано справо называется треугольником Паскаля — в n-ой строчке этого треугольника содержатся коэффициенты для разложения $(a+b)^n$. Число номер $k+1$ в n-ой строчке называется биномиальным коэффициентом C_n^k. Например,

$$C_3 : 0 = 1, \quad C_5^2 = 10, \quad C_4^2 = 6.$$

Эти числа возникают в задаче о числе сочетаний: C_n^k — это число способов быбрать k элементов и n различных. Например, чило байт, в которых ровно 3 единицы – это число C_8^3 — число способов выбрать три бита, в которых будут стоять единицы, из возьми бит байта.

Обратите внимание на то, что каждое число в треугольнике Паскаля равно сумме двух чисел из предыдущей строчки, которые находятся над ним.

З 2.1. Докажите, что

$$C_n^1 = n, \quad C_n^k + C_n^{k+1} = C_{n+1}^{k+1}, \quad C_n^k = \frac{n!}{k!(n-k!)}.$$

Рассмотрим две программы, которыерешают следующие задачи:
1) Запрограммировать функцию `C(n,k)` $= C_n^k$.
2) Вывести на экран треугольник Паскаля.

```
#include <stdio.h>
long C(long n,long k)
{
    if(k == 0 || n == k) return 1;
    return C(n - 1, k - 1) + C(n - 1, k);
}
int main()
{
    long n, k;
    scanf ("%ld%ld", &n, &k);
    printf ("%ld ", C(n, k));
    return 0;
}
```

Вопросы и задания

1. Сколько раз вызовется функция C(., .) при вычислении C(n, k)?
2. Докажите, что время вычисления $\frac{k}{n}$ по приведенному алгоритму пропорционально $\frac{k}{n}$.
3. Оцените ассимптотику $\frac{n/2}{n}$, а именно, напишите программу, которая вычисляет $\log C_n^{n/2}$ для $n = 2, 4, \ldots, 40$ и нарисуте график $\log C_n^{n/2}$ от n.

```
#include <stdio.h>
#define N 1000
long c[N];
int main ()
{
    long n, i, j;
    scanf ("%ld",&n);
    for(i = 1; i <= n ; i++) c[i] =0;
```

```
    c[0] = 1;
    for(j = 1 ; j <= n; j++)
        for(i = j; i >= 1 ; i--)
            c[i] = c[i-1] + c[i];

    for(i = 0; i <= n ; i++)
        printf ("%ld ", c[i]);

    return 0;
}
```

Вопросы и задания

1. Докажите, что указанный алгоритм вычисления n-ой строчки треугольника Паскаля работает быстрее, чем алгоритм вычисления C_n^k из предыдущей программы, а именно время работы пропорционально n^2.
2. Начиная с какого n самое большой число из n-ой строчки не умещается в тип **long**?

Корень уравнения

Рассмотрим уравнение, которое не имеет явной формулы для корня:

$$e^x = x - 2.$$

С помощью компьютера можно найти положительный корень этого уравнения с точностью до 10 знаков после запятой. Вот решение этой задачи:

```
#include <stdio.h>
#define EPS 1e-10
double f(double x)
{
    return exp(x) - 2 - x;
}
void main()
{
    double l = 0, r = 2;
    while( r - l < EPS )
    {
        c = ( l + r ) / 2;
        if( f(c) * f(r) < 0 ) l = c;
        else r = c;
    }
    printf ("%.10lf\n", (l + r)/2 );
}
```

Директива **#define EPS 1e-10** означает везде, где встречается комбинация EPS заменить её на число `1e-10`$= 1 \cdot 10^{-10}$. Число `EPS` — это точность, с которой мы хотим найти корень.

Алгоритм вычисления корня основан не методе деления попалам. А именно, пусть мы значем, что корень функции находится между $l = 0$ и $r = 2$. Найдем середину c отрезка $[l,\ r)$. Корень находится на одном из промежутков: либо на $[l,\ c)$, либо на $[,\ r)$, а именно

на том, значение функции на концах которого имеет разные знаки (вспомните теорему Ролля из курса мат. анализа). Выберем нужный из двух отрезков и применим к нему те же рассуждения. Будем заниматься делением попалам, пока размер отрезка не станет меньше необходимой точности.

Вопросы и задания

1. За один шаг длина отрезка $[l,\ r]$ уменьшается в два раза. Сколько нужно шагов, чтобы уменьшить отрезок в более чем 1000 раз?
2. Сколько требуется шагов, чтобы начиная с отрезка длины $r - l = 2$ дойти до отрезка длины меньше 10^{-10}? Сколько требуется шагов, чтобы найти корень с точностью до 100 знаков после запятой?
3. В случае деления попалам у нас есть нижняя и верхняя граница для значения корня. С каждым шагом эти границы сближаются. В методе Ньютона нахождения корня уравнения у нас имеется одно число x — текущее приближение корня. И следующее приближение получается по следующему алгоритму: находим точку на графике с абциссой x и проводим из неё касательную к графику; абцисса точки пересечения касательной с осью абцисс будет новым значением x. Так делается до тех пор, пока новое x отличается от старого на число меньше, чем $10^{-10}/2$. Реализуйте этот алгоритм. Для этого вам понадобится определить еще одну функцию, которая возвращает значение производной $f'(x) = (e^x - x - 2)' = e^x - 1$.

Задачи

З 2.2. Посмотрите на код и определите, что он делает.

```
#include <stdio.h>
void main ()
{
    int n;
    scanf ("%d", &n);
    while(n)
    {
        printf("%d", n%2);
        n /= 2;
    }
}
```

Глава 3

Алгоритмы и структуры данных

Быстрая сортировка

В первой части мы рассмотрели сортировку пузырьком Bouble Sort. Алгоритм bouble sort хорош тем, что пишется очень быстро: два оператора **for**, проверка условия и **swap** (обмен местами двух элементов), если условие выполнено. Зато работает он алгоритм квадратичное время, то есть время работы пропорционально квадрату от числа элементов в массиве.

Здесь мы рассмотрим другой, более эффективный алгоритм Quick Sort. Суть его можно разъяснить одним предложением: выбираем один из элементов массива x, раскидываем остальные элементы по правую и по левую сторону от этого элемента (в зависимости от того, меньше или больше x) и затем применяем тот же самый алгоритм к правой и левой части.

Как видите алгоритм рекурсивный — Quick Sort делит массив на две части, такие, что любой элемент первой части больше лиюо равен любому элементу второй части, а затем применяет самого себя к каждой из двух частей. В нашем примере функция `qsort` в качестве первого аргумента принимает три аргумента:

`a`: адрес первого элемента в массиве,

`lo`: индекс первого элемента кусочка,

`hi`: индекс последнего элемента кусочка.

```
#include <stdio.h>
#define N 1000
qsort( int* a, int lo, int hi )
{
  int h, l, p, t;
  if (lo < hi) {
     l = lo;
     h = hi;
     p = a[hi];
     do {
          while ((l < h) && (a[l] <= p)) l++;
          while ((h > l) && (a[h] >= p)) h--
          if (l < h) {
             t = a[l]; a[l] = a[h]; a[h] = t;
          }
     } while (l < h);

     t = a[l]; a[l] = a[hi]; a[hi] = t;
```

```
     qsort( a, lo, l-1 );
     qsort( a, l+1, hi );
  }
}
int main()
{
    int n, i;
    int a[N];
    scanf (“%d”, &n);
    for( i = 0 ; i < n ; i++ )
        scanf(“%d”, &a[i]);

    qsort(a, 0, n - 1);

    for( i = 0 ; i < n ; i++ )
        printf(“%d ”, a[i]);
}
```

Сортировка строк в алфавитном порядке

```
#include <stdio.h>
#include <string.h>
#define N 1000
#define M 50
int main()
{
    int n, i;
    char names[N][M];

    scanf(“%d”, &n);
    for(i = 0; i < n ; i++)
    {
        fgets(names[i], M, stdin);
    }

    qsort(names, n, sizeof(char[M]), strcmp);

    for(i = 0; i < n ; i++)
    {
        printf(“%s”, names[i]);
    }
    return 0;
}
```

Сортировка методом кучи

```
#include<stdio.h>
#define N 100

typedef struct
{
    int key;
    // int value;
} ITEM;

class HEAP
{
    public:
        ITEM h[N];
        int size;

        HEAP()
        {
            size = 0;
        }

        int add(ITEM x)
        {
            if(size == N-1) return 0;
            h[++size] = x;
            checkup (size);
            return 1;
        }

        int extract_min(ITEM *x)
        {
            if(size ==0) return 0;
            *x = h[1];
            h[1] = h[size--];
            checkdown (1);
            return 1;
        }

    private:
        void checkup(int c)
        {
            int p = c / 2;
            if(p == 0)return;
            if(h[p].key > h[c].key)
            {
                ITEM tmp;
                tmp = h[p]; h[p] = h[c]; h[c] = tmp;
                checkup (p);
            }
```

```
        }

        void checkdown(int p)
        {
            int c = 2*p;

            if(c > size) return;
            if(c + 1 <= size && h[c+1].key < h[c].key) c++;

            if(h[c].key < h[p].key)
            {
                ITEM tmp;
                tmp =  h[c]; h[c] = h[p]; h[p] = tmp;
                checkdown (c);
            }

        }
};

int main()
{
    HEAP heap;
    int n, i;
    ITEM x;

    scanf ("%d", &n);

    for(i = 0 ; i < n ; i++)
    {
        scanf ("%d", &x.key);
        if( !heap.add (x) ) break;
    }

    while( heap.extract_min (&x) )
    {
        printf ("%d ", x.key);
    }

    return 0;
}
```

Лабиринт

```
#include <stdio.h>
#define N 100

typedef struct {
    int x, y;     // координаты ячейки
```

```
    int d;       // номер шага, на котором мы добрались до ячейки
    } cell;

int main()
{
    int i, j, n;
    cell q[N*N];       // очередь
    cell start, end;   // начальная и конечная ячейка
    int b, e;          // указатели на начало и конец очереди
    char map[N][N];    // карта лабиринта
    int v[N][N];       // флаг посещения ячейки


    scanf("%d", &n);
    for(i = 0; i < n; i++)
       scanf("%s", map[i]);

    scanf("%d%d", &(start.x), &(start.y));
    scanf("%d%d", &(end.x), &(end.y));

    for(i = 0; i<n; i++)
        for(j = 0; j<n; j++) v[i][j] = 0;

    v[start.x][start.y] = 1;
    start.d = 1;
    q[0] = start;
    for(b = e = 0; b <= e ; b++)
    {
        cell a;
        int x, y, d;
        x = q[b].x;
        y = q[b].y;
        d = q[b].d;
#define check(xx, yy) if(xx >= 0 && yy >= 0 && xx < n && yy < n && !v[xx][yy] && map[x
        check (x-1, y);
        check (x, y-1);
        check (x+1, y);
        check (x, y+1);
        // v[x][y]=2;
        v[x][y] = d;
    }

    printf("%d\n", v[end.x][end.y] );
    getchar();
    return 0;
}
```

Калькулятор

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

int expr(void);

int term(){
    int ich = getchar();
    int v;

    if(ich >= '0' && ich <= '9' ){
        ungetc(ich, stdin);
        scanf("%d", &v);
        return v;
    }
    else if(ich == '-'){
        v = term();
        return -v;
    }
    else if(ich == '('){
        v = expr();
        assert(getchar() == ')');
        return v;
    }
    else
        assert(0);
    return -1;
}

int operation(int v){
    int ich = getchar();
    int v1;

    if(ich == '+'){
        v1 = term();
        return operation(v + v1);
    }
    else if(ich == '-'){
        v1 = term();
        return operation(v - v1);
    }
    else if(ich == '*'){
        v1 = term();
        return operation(v * v1);
    }
    else{
        ungetc(ich, stdin);
        return v;
    }
```

```
}

int expr(){
    return operation(term());
}

int main(){
    printf("%d\n", expr());
    return 0;
}
```

Корректный словарь

Корректный словарь — это словарь, корректный в метематическом смысле: некоторые понятия аксиоматические, а другие через них определяются. В корректном словаре не должно быть "циклических определений", в частности не должно быть такого, что A определяется через B, B определяется через C, а C определяется через A. Для каждого термина должно быть не больше одного определения. Те термины, для которых нет определения, считаются аксиоматическими.

Задача: написать программу, которая умеет определять корректный словарь или нет.

Вход. Первая строчка содержит натуральное число $D < 1000$ — количество словарных статей (определений). Затем идет D строчек с описанием этих определений. Каждое описание начинается с определяемого термина (слово без пробелов из латинских букв), затем идет число терминов (меньше 20), которые используются в определениие этого термина, и затем через пробел перечислены используемые в определении термины, разделенные пробелами. Все термины состоят менее чем из 25 букв.

Выход. Выход должен содержать либо "CORRECT", либо "NOT CORRECT" без кавычек.

Примеры работы программы:

Вход	Выход
2 aaa 3 bbb ccc ddd bbb 1 ccc	CORRECT
2 think 2 mind intellect mind 1 think	NOT CORRECT
2 aaa 1 a aaa 2 c d	NOT CORRECT

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

```
#include <string.h>
#define M 21
// max term width
#define N 1000
// maximum number of terms
#define NE 21
// maximum edges from one vertex
#define P 2011
// hash table size

/* ne[i] number of edges from i-th term  */
int ne[N];

/*  number of terms */
int n = 0;

/*  vertex color array */
int v[N];

/*  hash table */
int hash_table[P];

/*  terms */
char term[N][M];

/* lists of adjacent vertices for each vertex */
int a[N][NE];


/* returns id of the term (index of term in array term) */
int
getid ( char* s ){
    unsigned long i, h1 = 0, h2 = 0;

    /* calculate two hash values */
    for(i=0 ; s[i] ; i++)
        {
            h1 *= 13; h1 += s[i] % 13;
            h2 *= 17; h2 += s[i] % 17;
        }
    h1 %= P;
    h2 %= P-1; h2++; /* h2 should be > 0 and  < P */

    while( hash_table[h1] != -1)
    {
            /* collision or term s is already known  */
            if( strcmp ( term[hash_table[h1]], s ) == 0 ) return hash_table[h1];
            h1 += h2;
            h1 %= P;
```

```
    }

    /* we have got new term */
    hash_table[h1] = n;
    strcpy (term[n], s);
    return n++;
}

int check(int id){
    int i;
    if(v[id] == 1) return 0;
    v[id] = 1;
    for(i = 0; i < ne[id] ; i++)
        if( !check( a[id][i]) )return 0;
    v[id] = -1;
    return 1;
}

int main(){
    int i, j, D, m, id1, id2;
    char word[M];


    for(i = 0 ; i < P ; i++) hash_table[i] = -1;
    for(i = 0 ; i < N ; i++) v[i] = 0;

    scanf ("%d", &D);
    for(i = 0; i < D; i++){
        scanf("%s", word);
        id1 = getid (word);
        scanf("%d", &ne[id1]);
        if(v[id1] == 1 ){
            printf("NOT CORRECT\n");
            return 0;
        }
        v[id1] = 1;
        for(j = 0 ; j < ne[id1] ; j++){
            scanf ("%s", word);
            id2 = getid (word);
            a[id1][j] = id2;
        }

    }

    for(i = 0 ; i < n ; i++) v[i] = 0;
    for(i = 0 ; i < n ; i++)
            if( v[i] == 0 && !check (i) ) break;
    if(i == n) printf ("CORRECT\n");
    else printf ("NOT CORRECT\n");
```

```
    return 0;
}
```

Разбиение на отрезки

База данных

Нужно написать простейшую “базу данных”, в которой будут хранится фамилии людей и год их рождения. Необходимо, чтобы база данных быстро обрабатывала операцию добавления и извлечения данных и при этом могла содержать очень большое количество записей. Запись — это пара {фамилия → год рождения}). Будем считать, что все записи реализованны в виде структуры из двух элементов и занимают одинаковое место в памяти. Представьте себе, что всего в базе миллион записей.

Лобовое решение

Самое простое (но не самое лучшее) решение — это хранить записи в виде массива в некотором произвольном, неупорядоченном виде. Когда добавляется новая запись, она просто добавляется в конец массива — это делается очень быстро.

Теперь обратимся к операции поиска. Когда нам придет запрос “найти запись на Иванова”, нам придется идти по элементам массива и сравнивать фамилии с фамилией “Иванов”. В худшем случае, нам придется пробежаться по всему массиву и сделать, таким образом, миллион шагов (например, когда Иванова в базе данных нет, что совсем не редкий случай). Миллион операций — это очень дорого для такой простой операции.

Есть по крайней мере четыре подхода к решению этой проблемы.

Отсортированный массив

Улучшения можно добится, если хранить записи в упорядоченном по фамилии массиве (имеется ввиду алфавитный порядок).

Тогда, поиск можно осуществлять быстрее, а именно, методом деления попалам (см. задачу о поиске корня уравнения на стр. 18): берём элемент из середины массива; если фамилия в этой записи есть “Иванов”, то мы нашли искомый элемент. Если фамилия другая, то мы, можем определить, с какой стороны от этой записи должна находится запись на Иванова, смотрим с до или после найденной фамилии идет фамилия Иванова в алфовитном (лексографическом) порядка. После этого применяем ту же самую процедуру для новой части. И таким образом, за каждый шаг сужаем массив поиска в два раза.

Если мы ищем в массиве из миллиона элементов ($n = 1\,000\,000$), то операцию деления попалам нам нужно будет осуществлять не более, чем $[\log_2 n] = [\log_2 1\,000\,000] = 20$ раз.

Итак, когда записи отсортированны, мы можем выполнять операцию поиска за логарифмическое по количеству записей время.

Зато посмотрите, насколько сильно усложнилась операция добавления элемента: новый элемент мы должны добавить туда, где он должен находится в отсортированном массиве. Это значит, что за логарифмическое время мы найдем место, куда мы должна вставить. Потом нам, собственно, нужно осуществить вставку. В массиве элементы у нас хранятся друг за другом и поэтому нам нужно блок из всех элементов, которые должны следовать после вставляемого, переместить на одну ячейку вперед. Если вставляемый элемент попал в начало (идет первым по алфавиту), но нам передется передвинуть n записей, если в середину, то $n/2$, если в конец, то 0. В среднем получается $n/2$ — линейно по числу элементов время.

Получается, что если операции вставки и поиска осуществляются с одинаковой частотой, то мы не оптимизировали, а ухудшили работу нашей базы данных.

Нужно улучшить процедуру вставки — придумать какую-нибудь структуры, которая упорядочена, но при этом процедура вставки осуществляется быстрее чем за линейное время.

Следует сразу сказать, что поиск быстрее чем за $\log_2 n$ сделать просто невозможно, это следствие теории информации. Оказывается есть такие в некотором смысле упорядоченные (настолько упрядоченные, что можно поиск элемента осущствляется за время $\log_2 n$) структуры данных, где вставку элемента тоже можно осуществлять за время $\log_2 n$.

Индексирование – массив массивов

Давайте вместо одного массива заведем 26 больших.

В первом массиве находятся записи на фамилии, которые начинаются с буквы 'A', во втором — те, которые начинаются с буквы 'B', в последнем — те, которые начинаются с буквы 'Z'.

Таким образом, когда приходит запрос (добавляется новый элемент) мы срезу же узнаем в каком массиве искать (в какой массив добавлять). В среднем, надо ождать, что в каждом из массивов будет находится примерно $n/26$ элементов. Операция определения массива к которой относится операция довавления/посика осуществляется мгновенно, но при этом дает значительную оптимизацию — параметр n для каждого из массивов уменьшается в 26 раз.

Конечно, ориентироваться на первую букву неправильно. Потому что некоторые буквы как первые буквы более вероятны, чем другие. А некоторые буквы вообще не могут быть первой буквай фамилии. Правильное решение — вычислять hash функцию, которая принимает значения, скажем от 0 до 1000.

Определение 3.1. ***Hash-функция*** *— функция, принимающаяя в качестве аргумента некоторый элемент (запись базы данных, строчку, текст, файл картинки, ...) и возвряющаюшая чило, обычно целое число. Важно, чтобы hash-функция возвращала самые разные значения и чтобы эти значения были равновероятны, то есть чтобы на множестве значений hash-функции было как можно более равномерное распределение.*

Равномерность распределения значений hash-функции полезна в данном случае, поскольку нам нужно чтобы в каждом из 26 массивов было примерно однинаковое количество элементов. Если все элементы соберутся в одном из 26 массивов, то никакой оптимизации мы не получим. И как, мы уже отметили, лучше когда массивов будет не 26 а 1000 или около того.

Уж очень не хочется выделять память под $M = 26$ и тем более $M = 1000$ массивов длиной милллион. Если запись состаящая из фамилии и целого числа занимает 300 байт, то нам потребуется

$$M \cdot 1\,000\,000 \cdot 30/10^6 = M \cdot 30 \text{ Mbyte}.$$

Даже один массив выделять жалко — максимальное возможное количество элементов может сильно превосходить среднее значение элементов, и мы будет не эффективно использовать память компьютера.

Нас может спасти динамическое выделение памяти.

Динамическое выделение памяти, списки и двойное индексирование

Определение 3.2. ***Динамическое выделение памяти*** *— это выделение памяти во время выполнения программы. Когда программа работает, она знает какое число элементов реально присутствует в базе и выделяет для них ровно столько памяти, сколько нужно*

(точнее несколько больше). Когда лимит памяти исчерпывается, нужно запускать команду повторного выделения памяти `realloc` *(re-allocate memory). При этом, если есть возможность, происходить просто перемещение верхней границы массива, а если это невозможно (за нашим массивом в памяти идут кусочки занятые другими данными), то находится новое пустое место нужного расзмера и наш массив целиком туда перемещается.*

Но индексирование, как вы понимаете просто уменьшает параметр n в M раз, саму асимптотику (зависимость времени выполнения операции от n) оно не меняет. Если в каждом из массивов мы используем технику отсортированного массива, толь время добавления нового элемента будет по прежнему линейно зависеть n, хоть и уменьшится в n раз.

Идею динамического программирования можно применять к более мелким единицам — к самим записям, и для каждой записи отдельно выделять небольшой кусочек памяти, ровно столько сколько нужно. Тогда они будут хаотически разбросаны в памяти комапьютера. Но для того, находить их и осуществлять поиск, необходимо связать их друг с другом. Например, выстраивать списки – в каждой записи, кроме самих данных, хранится еще ссылка на следующий в списке элемент (место в памяти, где начинаются данные относящиеся к следующей в списке). Такой подход исключает операции повторного выделения памяти (re-allocation), каждая запись навсегда остается в том месте памяти, куда они изначальна помещена.

Таким образом, мы можем содержать M списков, в списке с номером i находятся элементы, укоторых hash-функция равна i. То есть заводим массив размера M, i-ый элемент которого есть первый элемент в i-ом списке.

Проблема в том, что в списках, даже если они отсортированы невозможно осуществлять поиск методом деления попалам — мы не знаем какой элемент находится в середине списка и не можем по двум элементам списка найти элемент, который находится посередине между ними.

Зато списки дают нам возможность сделать еще один шаг — осуществить двойное индексирования. Давайте заведем две разные hash-функции — это может быть, например, первая буква и вторая буква фамилии. И организуем в памяти не массив из M списков, а массив из M массивов, в каждый из которых содержит массив из M списков.

Далее естественным образом возникает идея тройного индексирования и становится понятно, что нам нужна другая структура данных — дерево.

Дерево поиска

Дерево поиска соответствует индексированию до конца. Фамилии естественным образом выстраиваются в дререво.

Чтобы написать структуру деревы, достаточно описать один элемент дерева (узел дерева) и определить функции добавления, удаления и поиска элемента.

Hash-таблица

Поиск кратчайшего пути

Глава 4

Другие языки программирования

Perl

```
    print sort <>;

# dictionary.pl

use strict;
# map word  -> list of words it is derived from
my %DEF;
# max word -> color of the word (is this word visited or not)
my %V;

sub check {
    my $term = shift;
    return 0 if $V{$term} == 1;
    $V{$term} = 1;
    foreach my $w (@{$DEF{$term}} ) {
        return 0 if  not check ( $w );
    }
    $V{$term} = -1;
    return 1;
}

print "Enter number of words: ";
my $D = <>;
chomp $D;
for ( my $i = 0; $i < $D; $i++ ) {
    print "Word " , $i+1 , ": ";
    my $term = <>;
    chomp $term;
    print "Число слов в определении термина $term: ";
    my $ne = <>;
    chomp $ne;
    die "NOT CORRECT\n" if $DEF{$term}; # повторное определение
    for( my $j = 0 ; $j < $ne; $j++ ) {
        print " слово " , $j + 1 , ": ";
        my $w = <>;
        chomp $w;
        push @{$DEF{$term}}, $w;
    }
```

```
}
foreach my $w (keys %DEF) {
    next if $V{$w};
    die "NOT CORRECT\n" if not check ( $w );
}
print "CORRECT\n";
```

Pyhton

TCL

Haskell