1.Массивы

Массив — это непрерывный участок памяти, содержащий последовательность объектов одинакового типа, обозначаемый одним именем. Массив характеризуется следующими основными понятиями: Элемент массива (значение элемента массива) – значение, хранящееся в определенной ячейке памяти, расположенной в пределах массива, а также адрес этой ячейки памяти. Каждый элемент массива характеризуется тремя величинами:

\* адресом элемента — адресом начальной ячейки памяти, в которой расположен этот элемент;

\* индексом элемента (порядковым номером элемента в массиве);

\* значением элемента.

Адрес массива – адрес начального элемента массива. Имя массива – идентификатор, используемый для обращения к элементам массива. Размер массива – количество элементов массива Размер элемента – количество байт, занимаемых одним элементом массива. Графически расположение массива в памяти компьютера можно представить в виде непрерывной ленты адресов.

Представленный на рисунке массив содержит q элементов с индексами от 0 до q-1. Каждый элемент занимает в памяти компьютера k байт, причем расположение элементов в памяти последовательное. Адреса i-го элемента массива имеет значение

n+k·i

Адрес массива представляет собой адрес начального (нулевого) элемента массива. Для обращения к элементам массива используется порядковый номер (индекс) элемента, начальное значение которого равно 0. Так, если

массив содержит q элементов, то индексы элементов массива меняются в пределах от 0до q-1. Длина массива – количество байт, отводимое в памяти для хранения всех элементов массива.

ДлинаМассива = РазмерЭлемента \* КоличествоЭлементов

Для определения размера элемента массива может использоваться функция

int sizeof(тип);

Например,

sizeof(char) = 1; sizeof(int) = 4; sizeof(float) = 4; sizeof(double) = 8;

Объявление и инициализация массивов

Для объявления массива в языке Си используется следующий синтаксис:

тип имя[размерность]={инициализация};

Инициализация представляет собой набор начальных значений элементов массива, указанных в фигурных скобках, и разделенных запятыми.

int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}; // массив a из 10 целых чисел

Если количество инициализирующих значений, указанных в фигурных скобках, меньше, чем количество элементов массива, указанное в квадратных скобках, то все оставшиеся элементы в массиве (для которых не хватило инициализирующих значений) будут равны нулю. Это свойство удобно использовать для задания нулевых значений всем элементам массива.

int b[10] = {0}; // массив b из 10 элементов, инициализированных 0

Если массив проинициализирован при объявлении, то константные начальные значения его элементов указываются через запятую в фигурных

скобках. В этом случае количество элементов в квадратных скобках может быть опущено.

int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};

При обращении к элементам массива индекс требуемого элемента указывается в квадратных скобках [].

Однако часто требуется задавать значения элементов массива в процессе выполнения программы. При этом используется объявление массива без инициализации. В таком случае указание количества элементов в квадратных скобках обязательно.

int a[10];

Многомерные массивы

В языке Си могут быть также объявлены многомерные массивы. Отличие многомерного массива от одномерного состоит в том, что в одномерном массиве положение элемента определяется одним индексом, а в многомерном — несколькими. Примером многомерного массива является матрица. Общая форма объявления многомерного массива

тип имя[размерность1][размерность2]...[размерностьm];

Элементы многомерного массива располагаются в последовательных ячейках оперативной памяти по возрастанию адресов. В памяти компьютера элементы многомерного массива располагаются подряд, например массив, имеющий 2 строки и 3 столбца,

int a[2][3];

будет расположен в памяти следующим образом

Общее количество элементов в приведенном двумерном массиве определится как

КоличествоСтрок \* КоличествоСтолбцов = 2 \* 3 = 6.

Количество байт памяти, требуемых для размещения массива, определится как

КоличествоЭлементов \* РазмерЭлемента = 6 \* 4 = 24 байта.

Инициализация многомерных массивов

Значения элементов многомерного массива, как и в одномерном случае, могут быть заданы константными значениями при объявлении, заключенными в фигурные скобки {}. Однако в этом случае указание количества элементов в строках и столбцах должно быть обязательно указано в квадратных скобках [].

Однако чаще требуется вводить значения элементов многомерного массива в процессе выполнения программы. С этой целью удобно использовать вложенный параметрический цикл.

Передача массива в функцию

Обработку массивов удобно организовывать с помощью специальных функций. Для обработки массива в качестве аргументов функции необходимо передать

\* адрес массива,

\* размер массива.

Исключение составляют функции обработки строк, в которые достаточно передать только адрес. При передаче переменные в качестве аргументов функции данные передаются как копии. Это означает, что если внутри функции произойдет изменение значения параметра, то это никак не повлияет на его значение внутри вызывающей функции. Если в функцию передается адрес переменной (или адрес массива), то все операции, выполняемые в функции с данными, находящимися в пределах видимости указанного адреса, производятся над оригиналом данных, поэтому исходный массив (или значение переменной) может быть изменено вызываемой функцией.

Вопрос № 2

В языке Си, структура (struct) — композитный тип данных, инкапсулирующий без сокрытия набор значений различных типов. Порядок размещения значений в памяти задаётся при определении типа и сохраняется на протяжении времени жизни объектов, что даёт возможность косвенного доступа (например, через указатели)

Синтаксис объявления структуры

|  |
| --- |
| struct <имя> {      <тип1> <поле1>;      <тип2> <поле2>;      ...      <типN> <полеN>;  }; |

Полями структуры могут быть любые объявленные типы, кроме самой структуры этого же типа, но можно хранить указатель на структуру этого типа:

|  |
| --- |
| struct node {      void\* value;      struct node next;  }; |

Нельзя, нужно

|  |
| --- |
| struct node {      void\* value;      struct node \*next;  }; |

**3. Указатели**

***Указатель*** – это переменная, которая хранит адрес области памяти. Указатель, как и переменная, имеет тип. Синтаксис объявления указателей

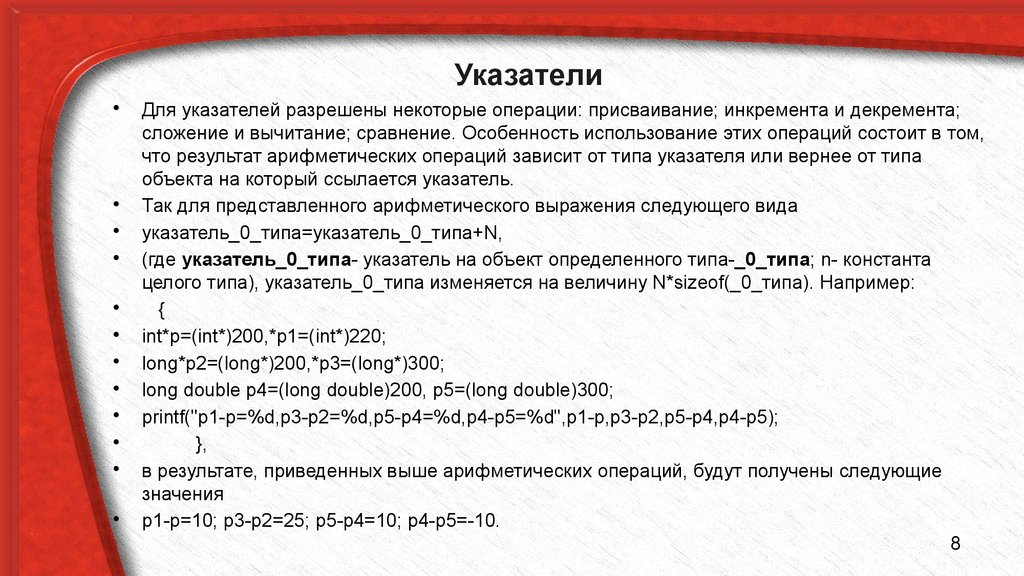
|  |
| --- |
| <тип> \*<имя>; |

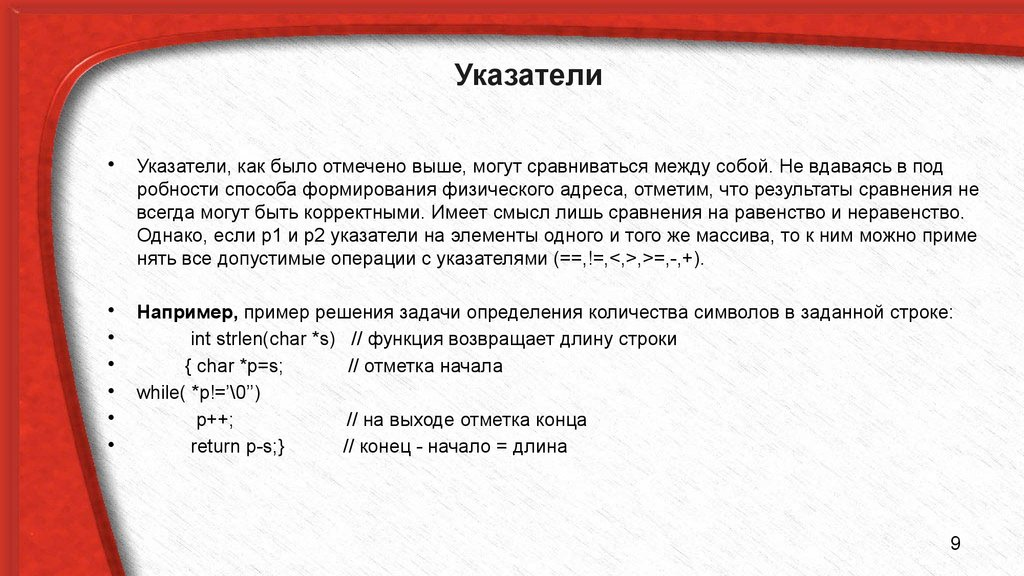
Например

float \*a;

long long \*b;

Два основных оператора для работы с указателями – это оператор & взятия адреса, и оператор \* разыменования.





**4. Объединения**

Объединениями называют сложный тип данных, позволяющий размещать в одном и том же месте оперативной памяти данные различных типов.

Размер оперативной памяти, требуемый для хранения объединений, определяется размером памяти, необходимым для размещения данных того типа, который требует максимального количества байт.

Когда используется элемент меньшей длины, чем наиболее длинный элемент объединения, то этот элемент использует только часть отведенной памяти. Все элементы объединения хранятся в одной и той же области памяти, начиная с одного адреса.

Общая форма объявления объединения

|  |  |
| --- | --- |
| **union ИмяОбъединения {   тип ИмяОбъекта1;   тип ИмяОбъекта2;   . . .   тип ИмяОбъектаn; };** | Объединение |

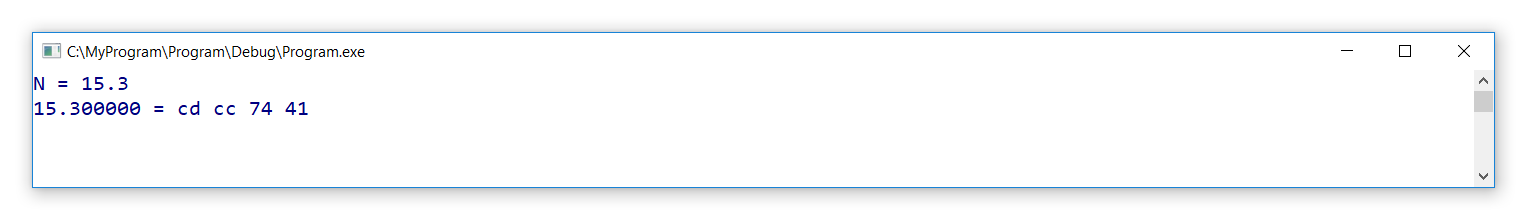
Объединения применяются для следующих целей:

* для инициализации объекта, если в каждый момент времени только один из многих объектов является активным;
* для интерпретации представления одного типа данных в виде другого типа.

**Пример программы, которую могут попросить написать на экзамене:**

Например, удобно использовать объединения, когда необходимо вещественное число типа float представить в виде совокупности байтов

#define \_CRT\_SECURE\_NO\_WARNINGS  
#include <stdio.h>  
#include <stdlib.h>  
union types  
{  
  float f;  
  unsigned char b[4];  
};  
int main()  
{  
  types value;  
  printf("N = ");  
  scanf("%f", &value.f);  
  printf("%f = %x %x %x %x", value.f, value.b[0], value.b[1], value.b[2], value.b[3]);  
  getchar();  
  getchar();  
  return 0;  
}

Результат выполнения: 

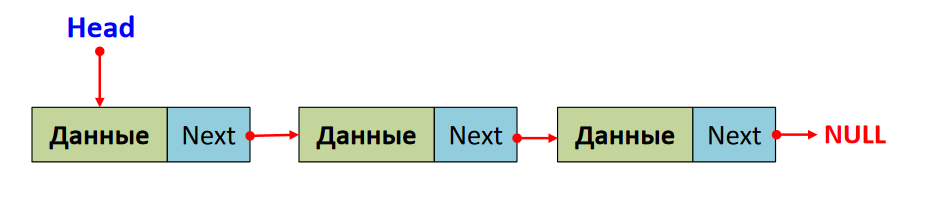
**5. Связанные списки**

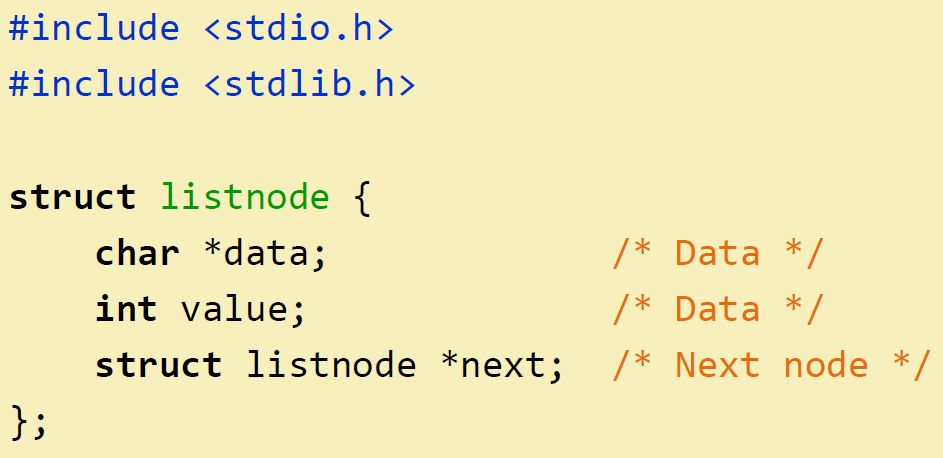
**Связный список (linked list)** – динамическая структура данных для хранения информации, в которой каждый элемент хранит указатели на один или несколько других элементов

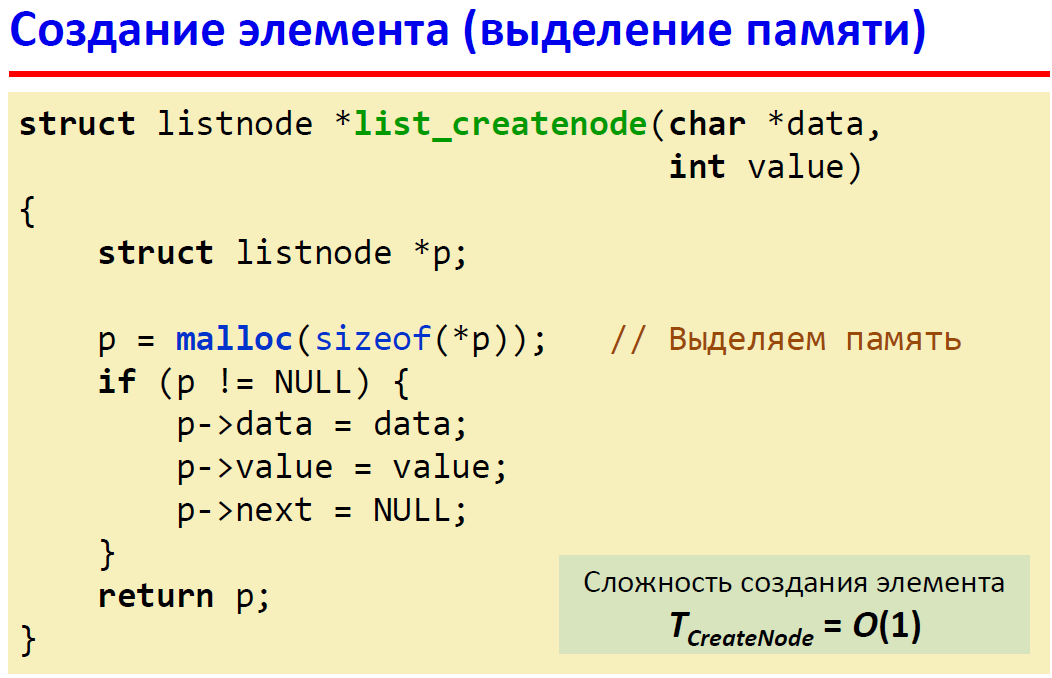
****

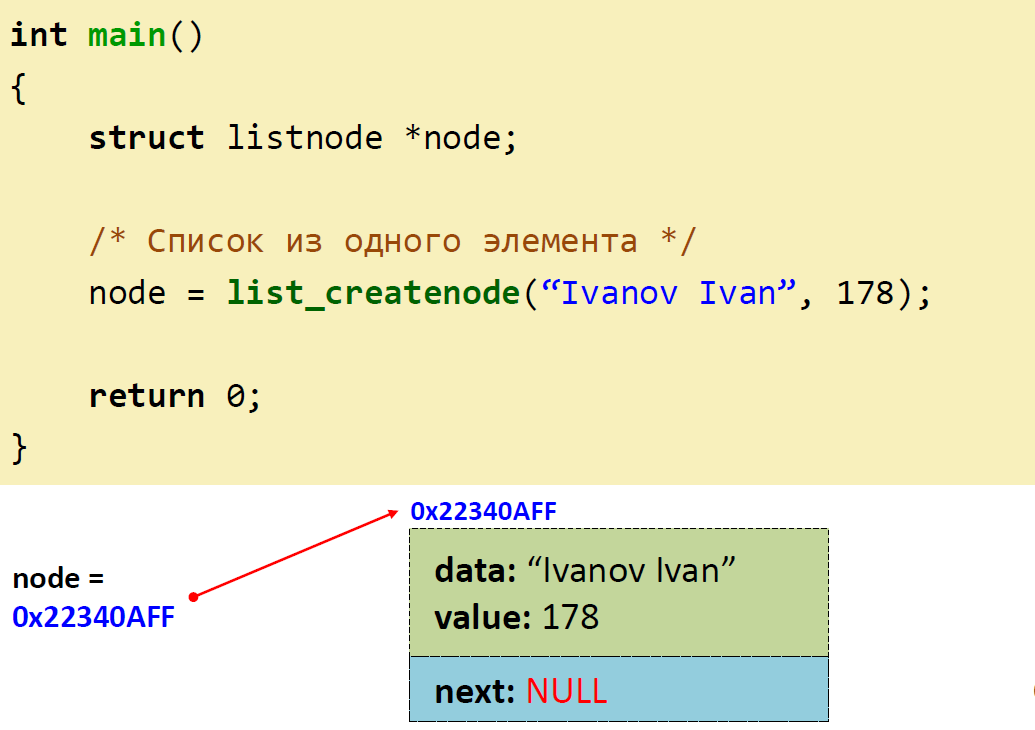
♣ Размер списка заранее не известен – элементы добавляются во время работы программы (динамически)

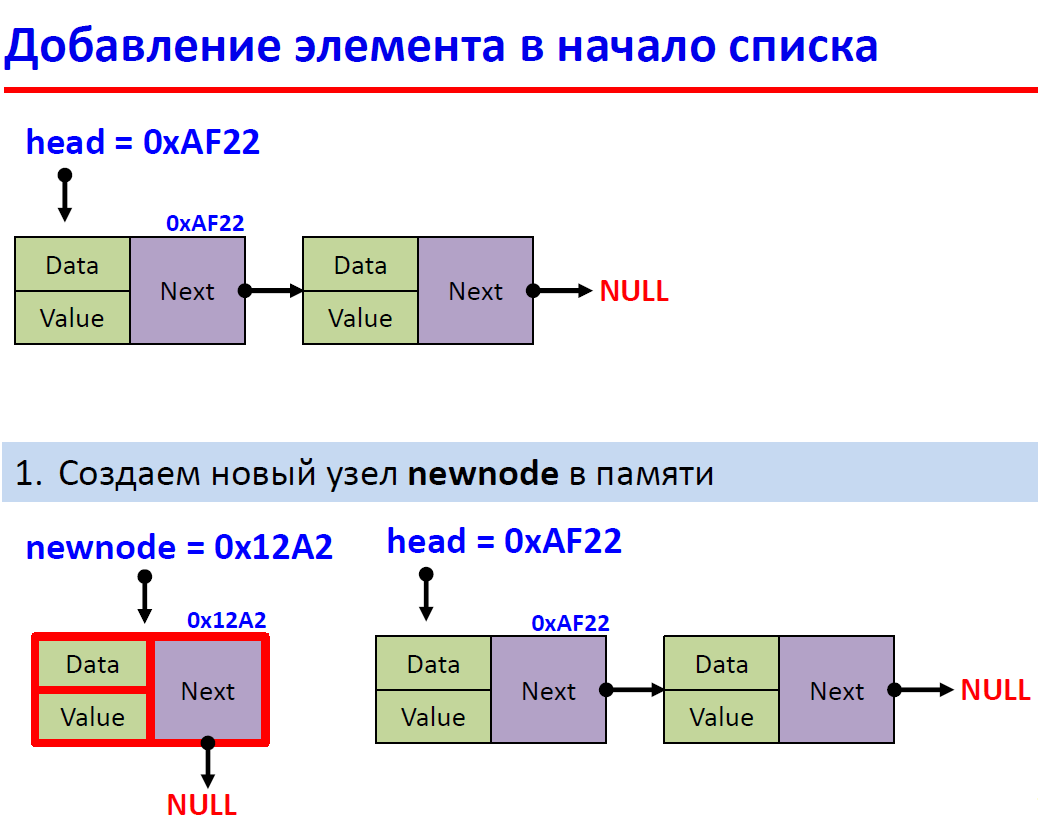
♣ Память под элементы выделяется динамически (функции: malloc, calloc, free)

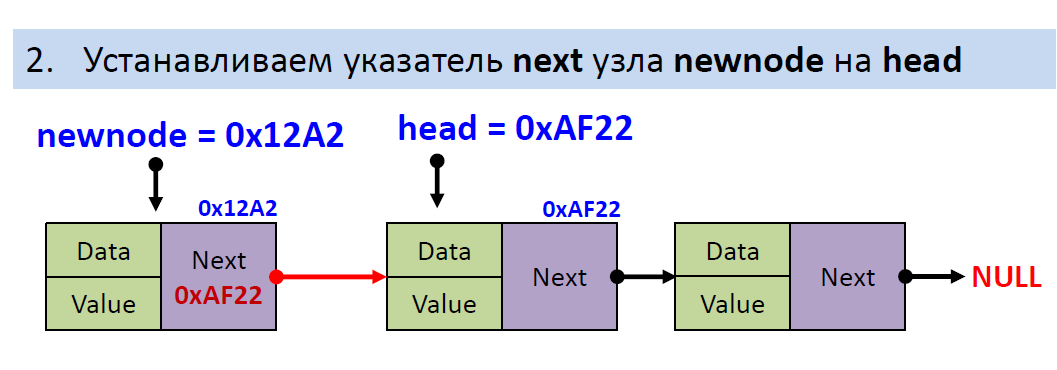
****

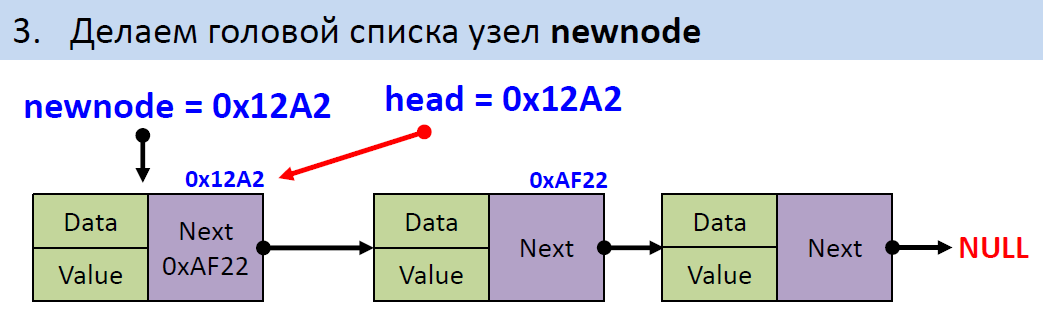
****

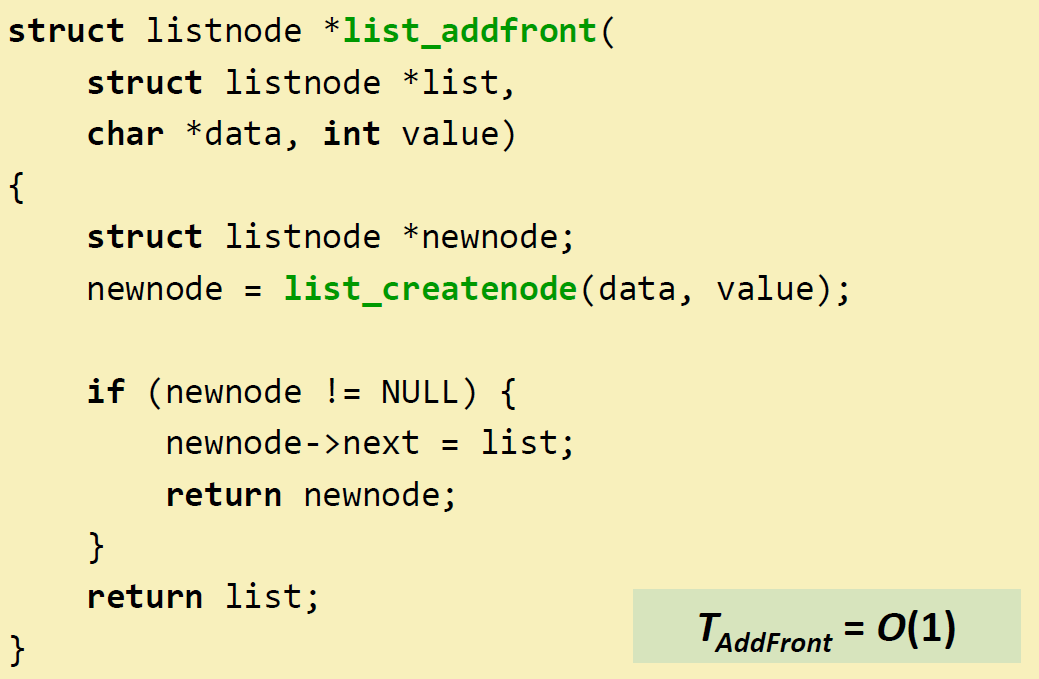
****

****

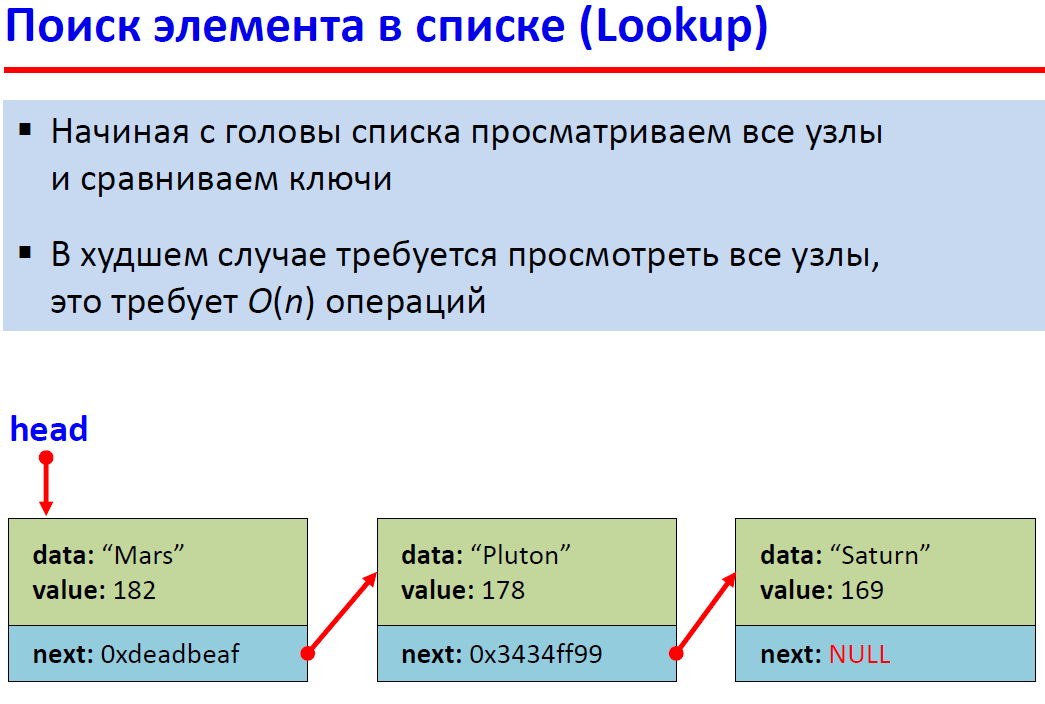
****

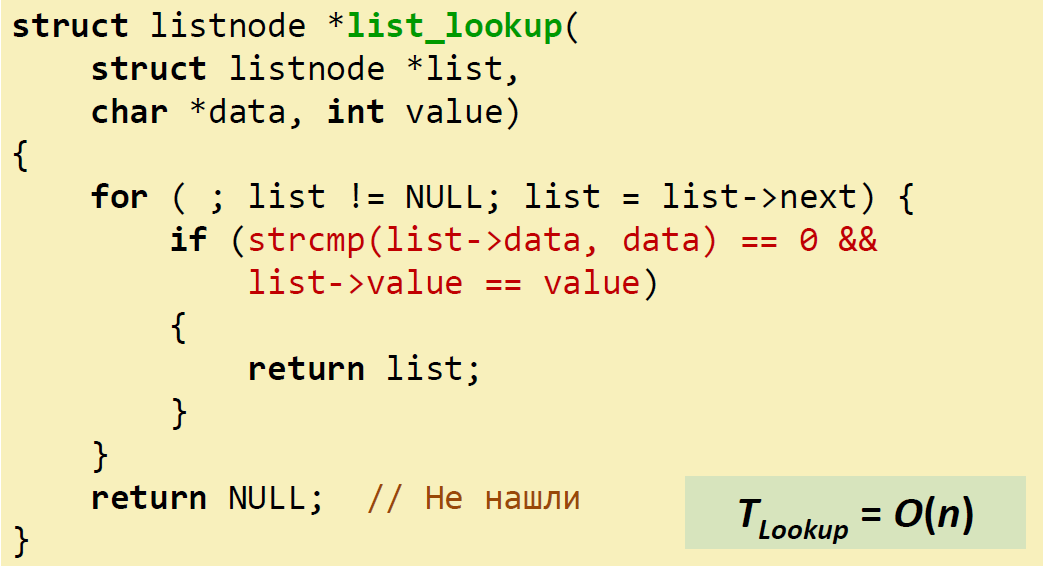
****

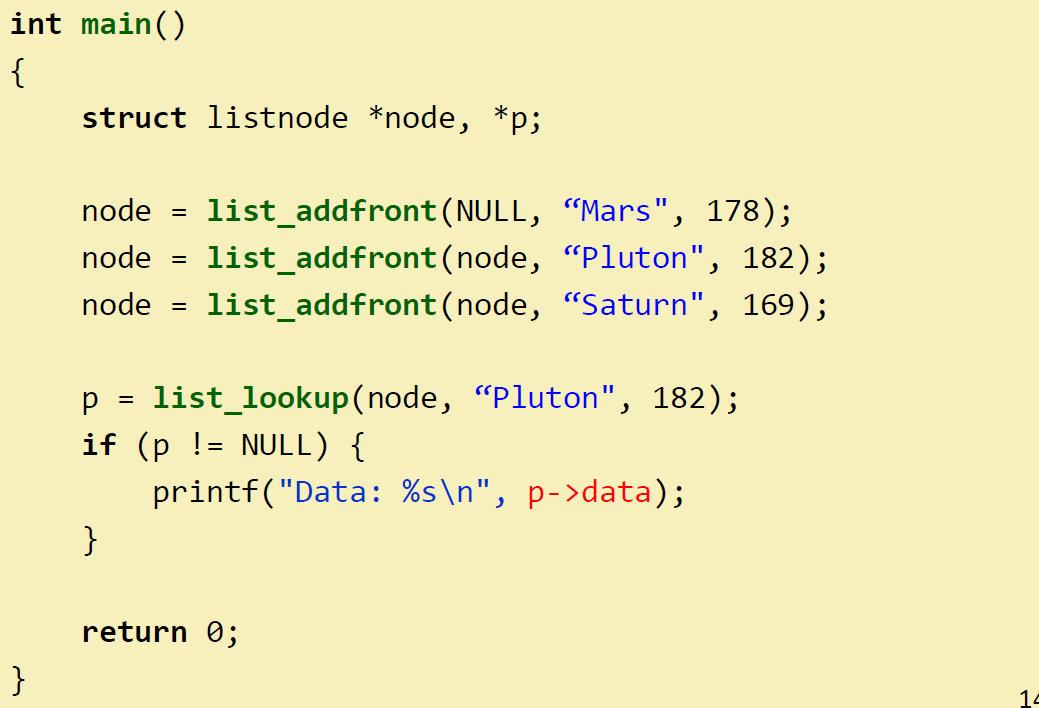
****

****

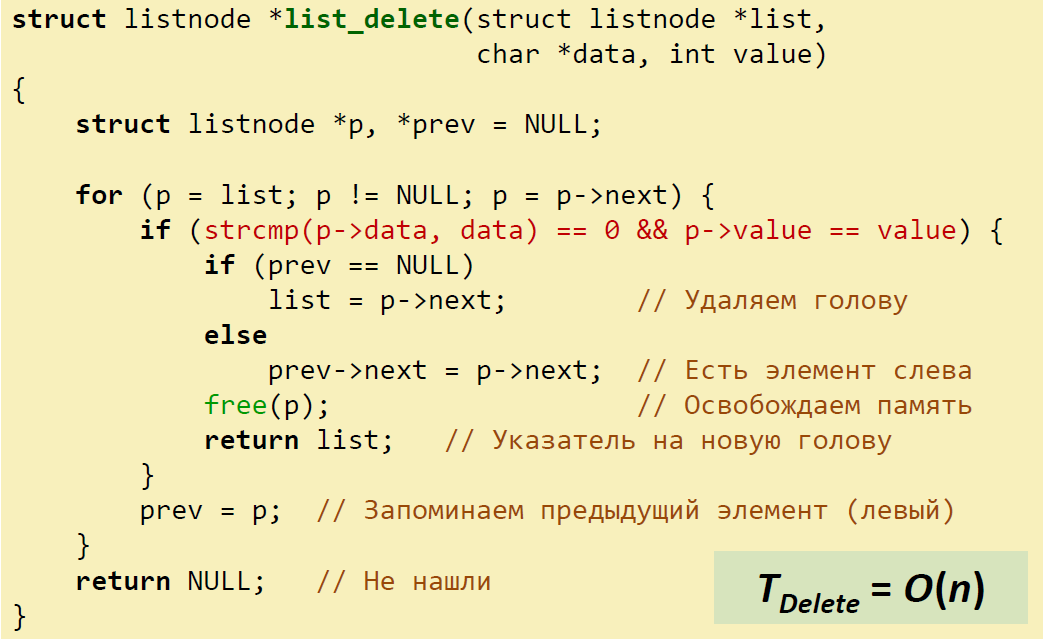
****

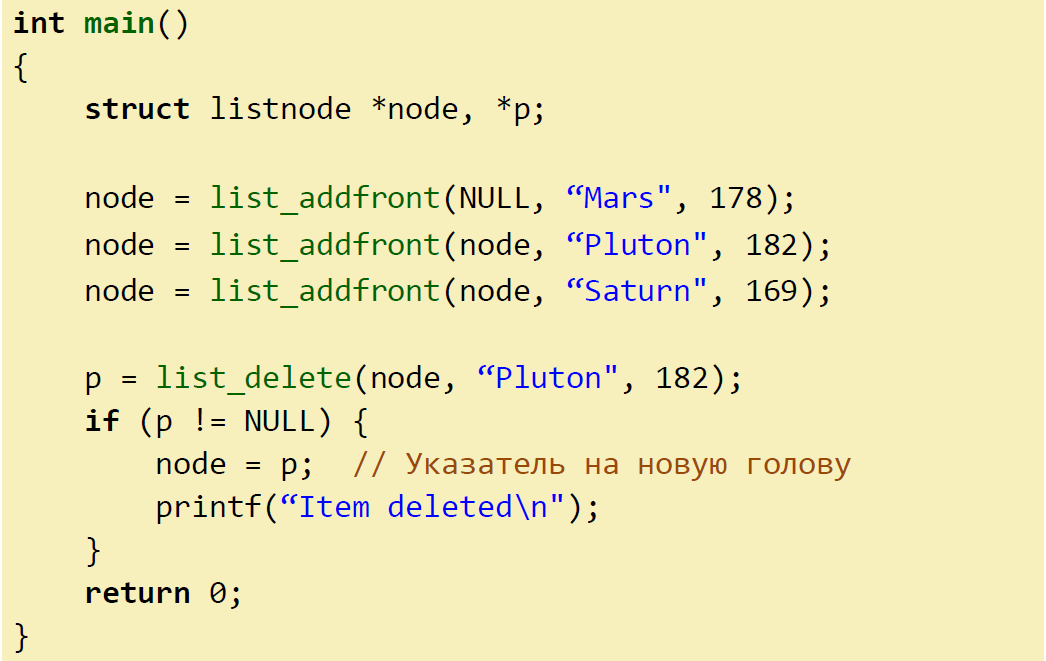
****

****

****

****

****

****

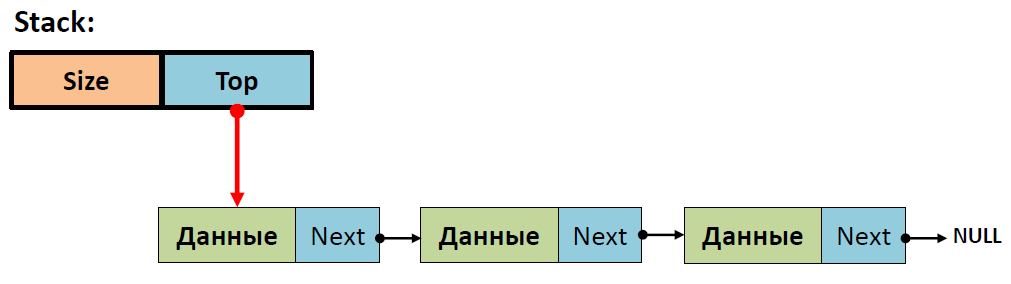
**6. Стек**

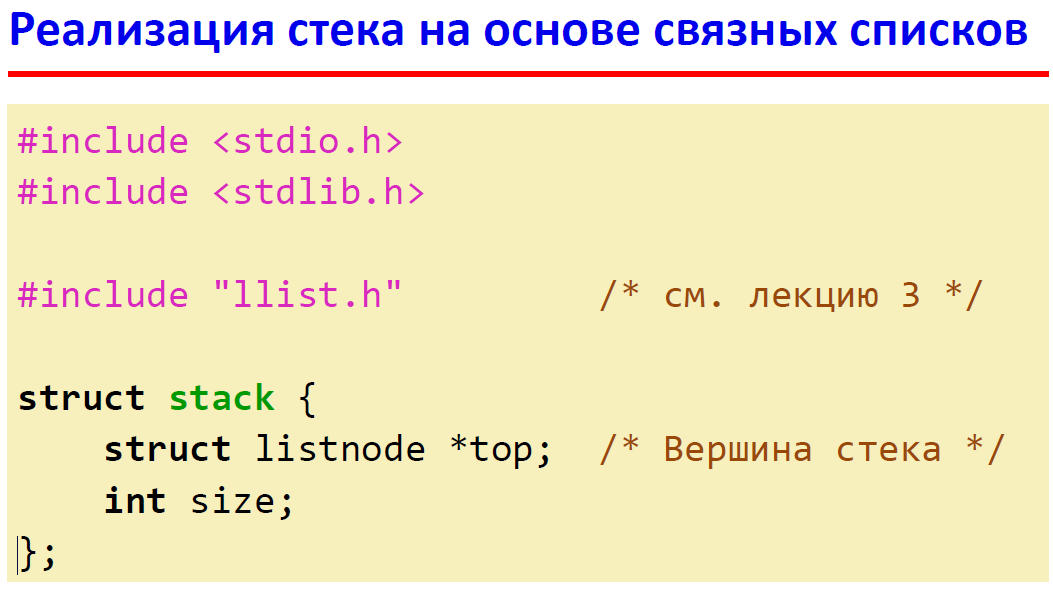
|  |  |
| --- | --- |
| **Стек (Stack)** –структура данных для хранения элементов  Дисциплина доступа к элементам: “последним пришел – первым вышел”  (Last In –First Out, LIFO)  Элементы помещаются и извлекаются с головы стека(top)  **Подходы к реализации стека**  **1.На основе связных списков** (linked lists)Длина стека ограничена объемом доступной памяти  **2.На основе статических массивов**  Длина стека фиксирована (задана его максимальная длина –количество элементов в массиве) |  |

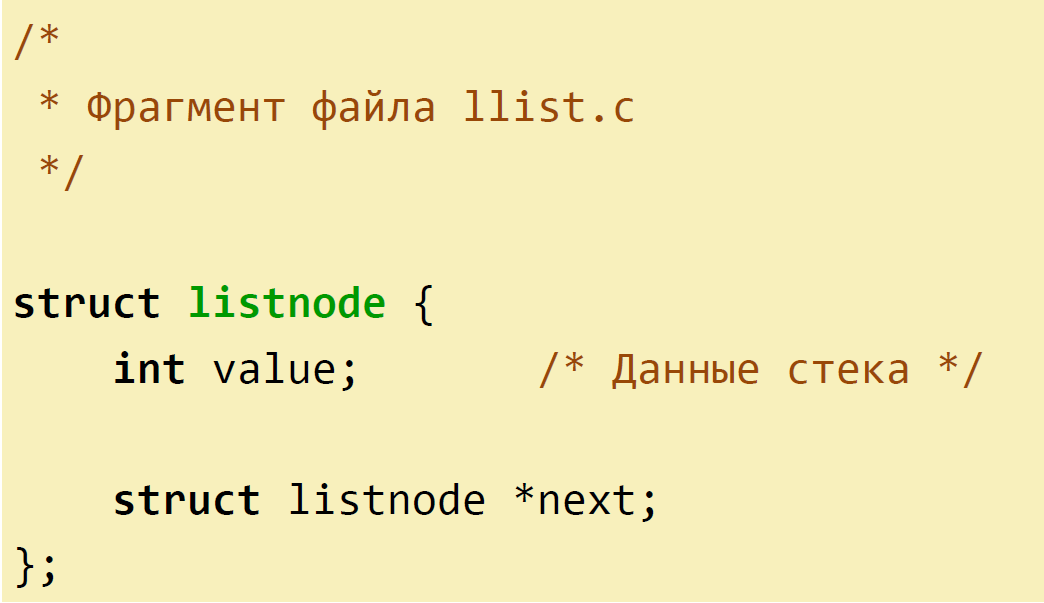
**Реализация стека на основе связных списков**

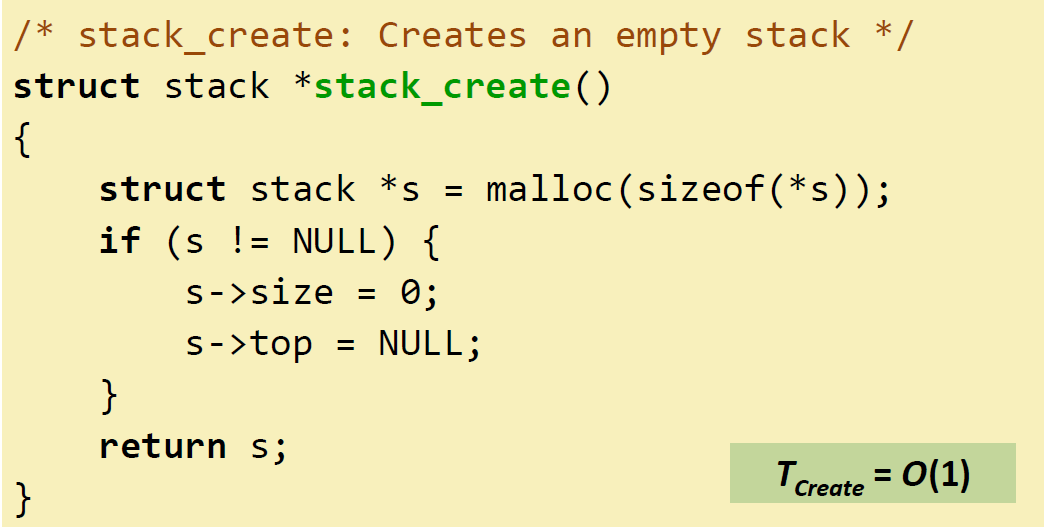
Элементы стека хранятся в односвязном списке (singly linked list)

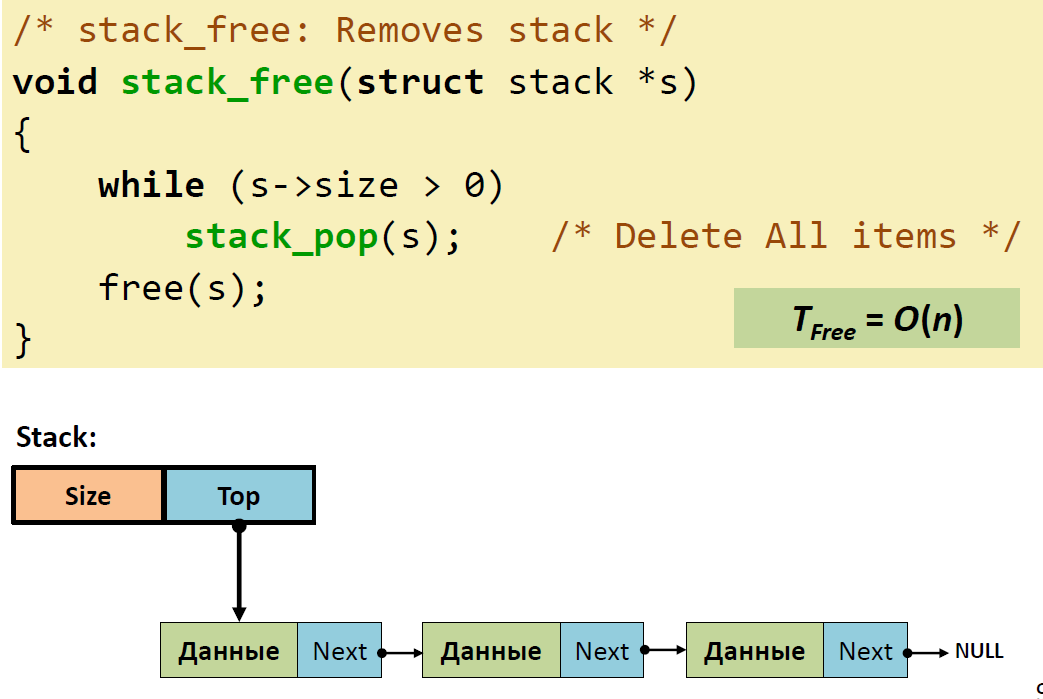
Добавление элемент и удаление выполняется за время *O*(1)

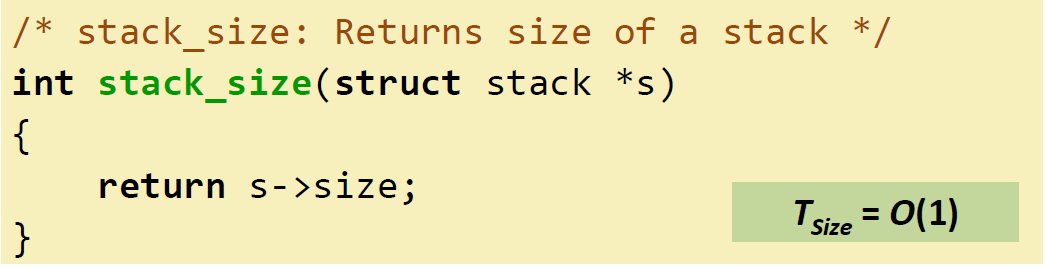
****

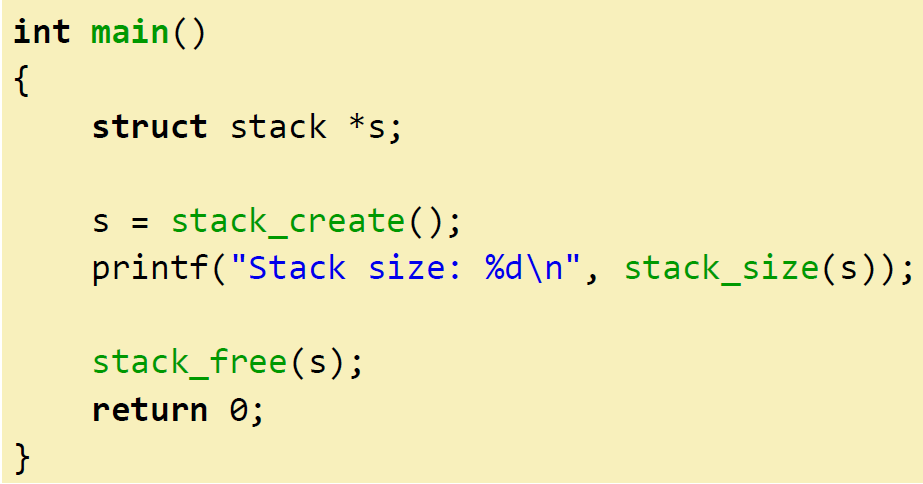
****

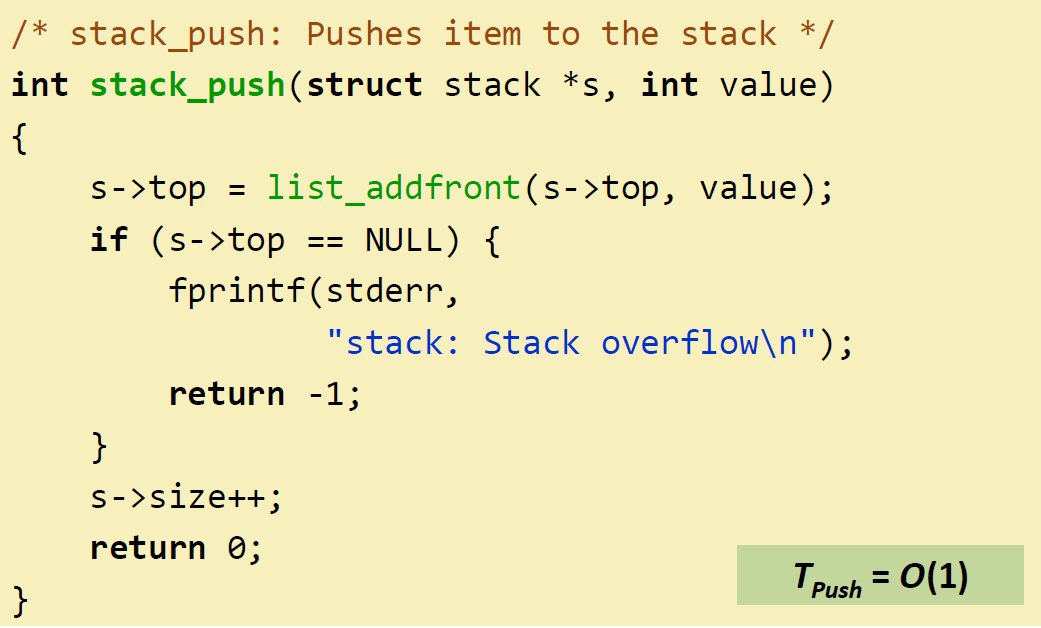
****

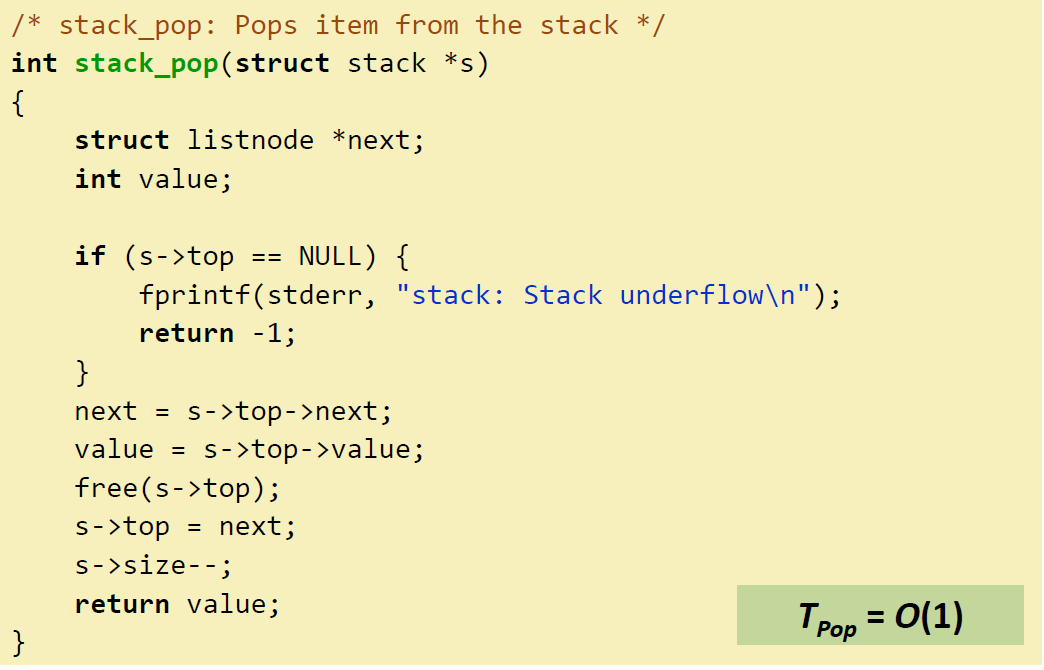
****

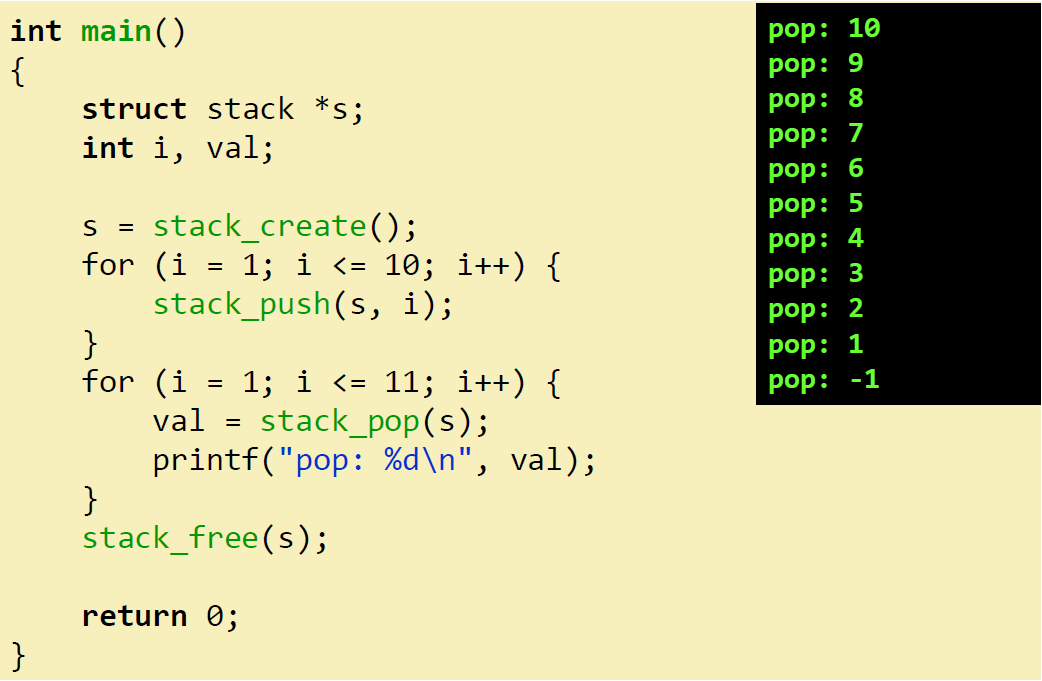
****

****

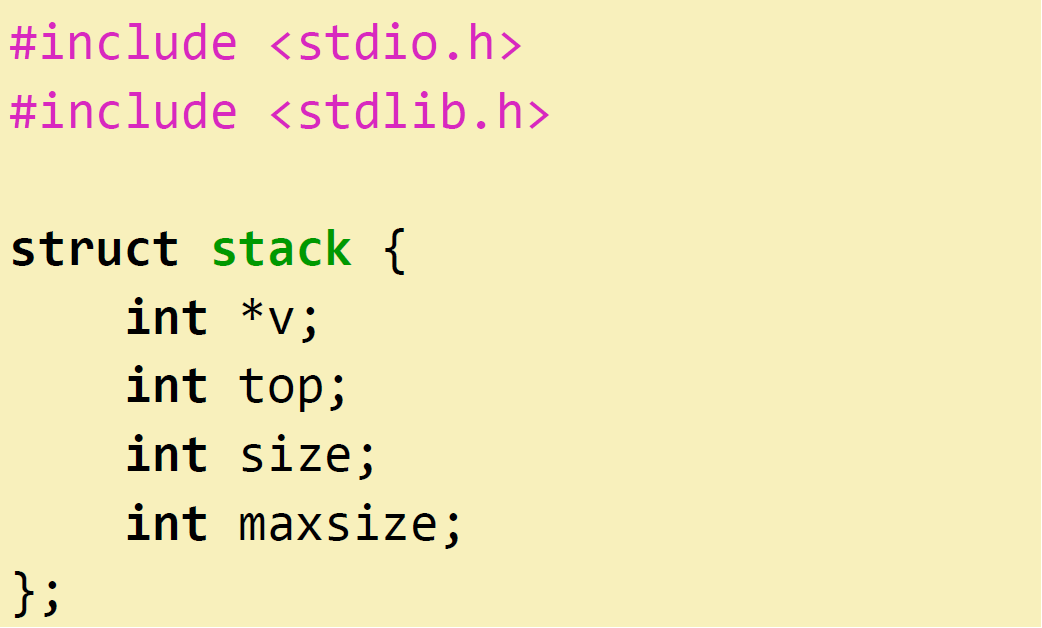
****

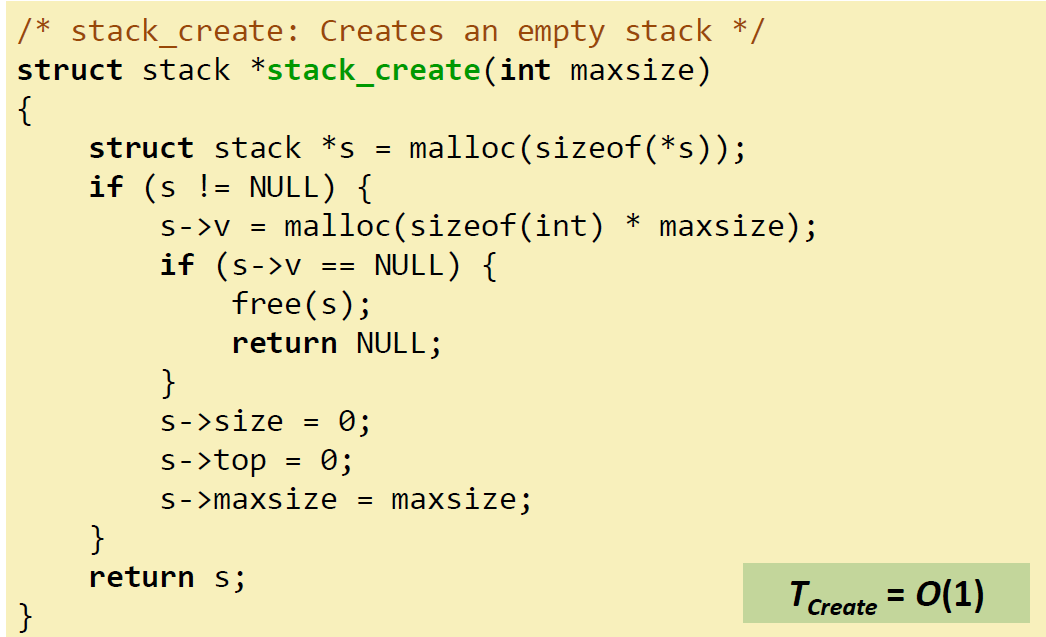
****

****

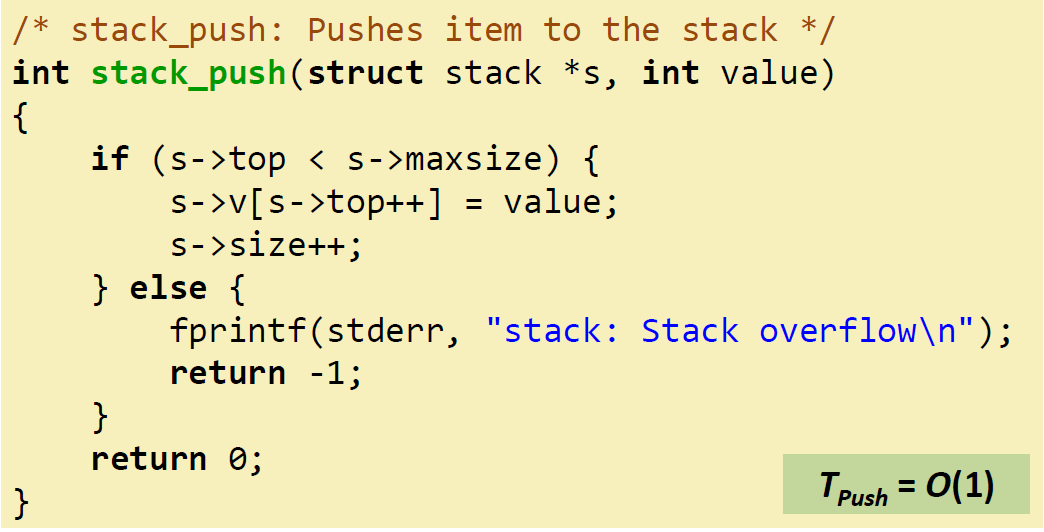
****

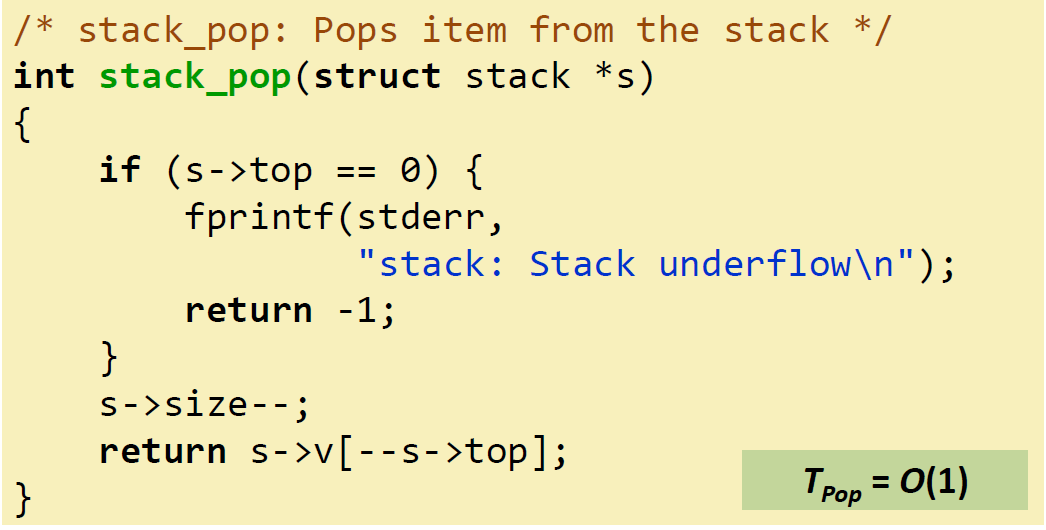
****

****

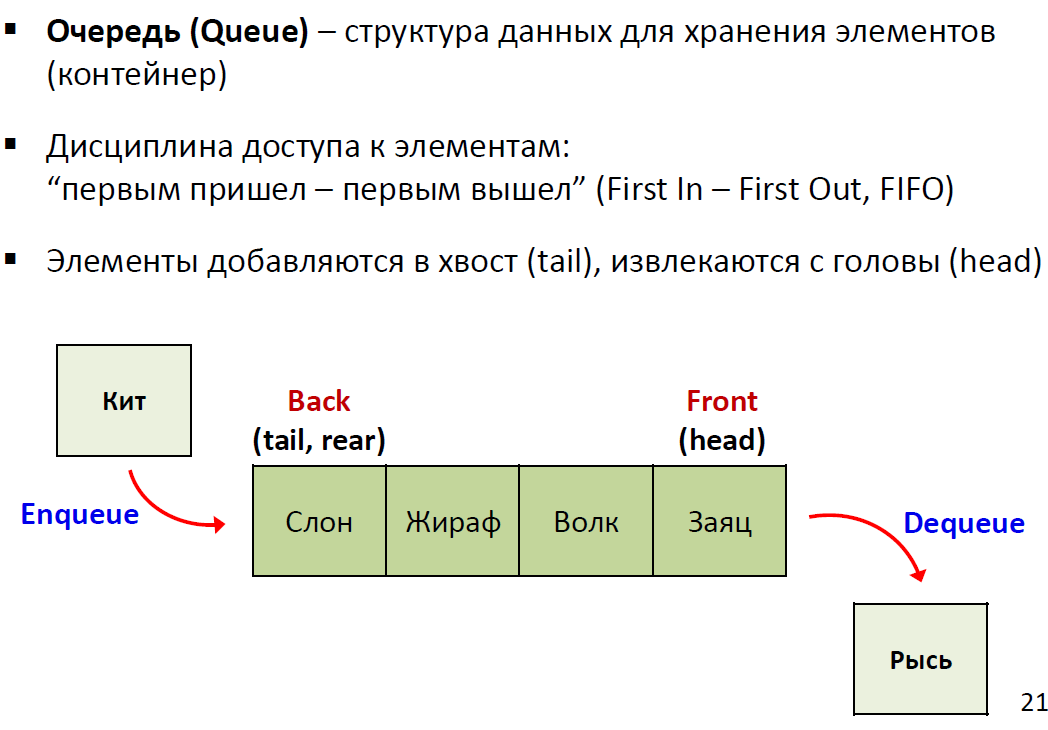
****

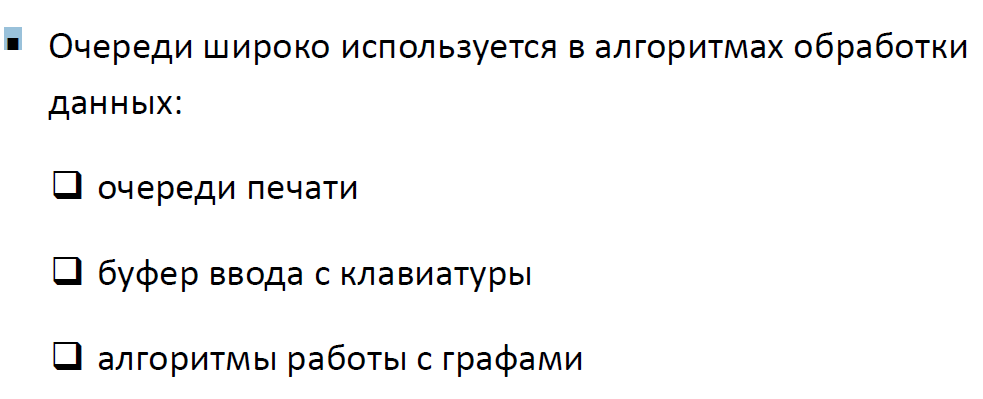
****

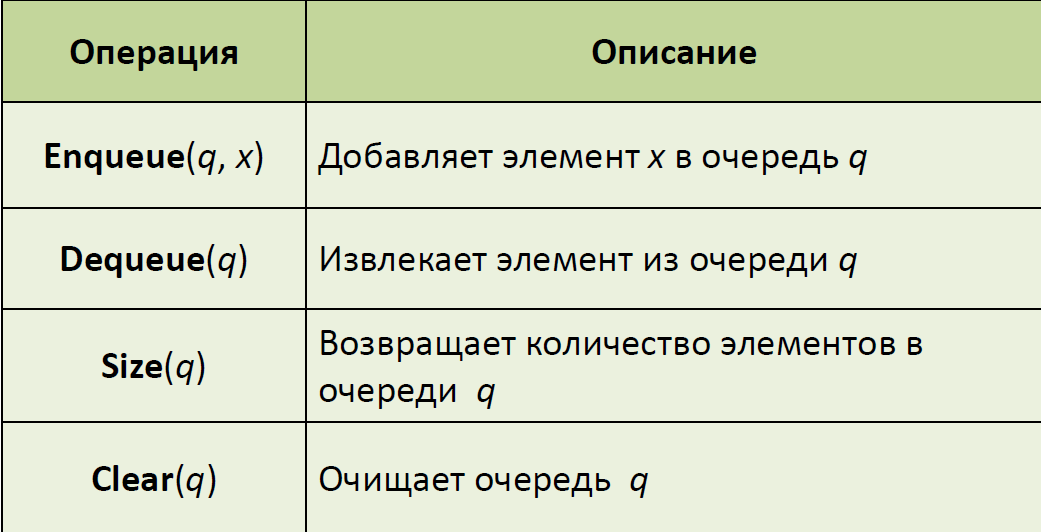
****

****

**7. Очередь**

****

****

****

**Подходы к реализации очереди**

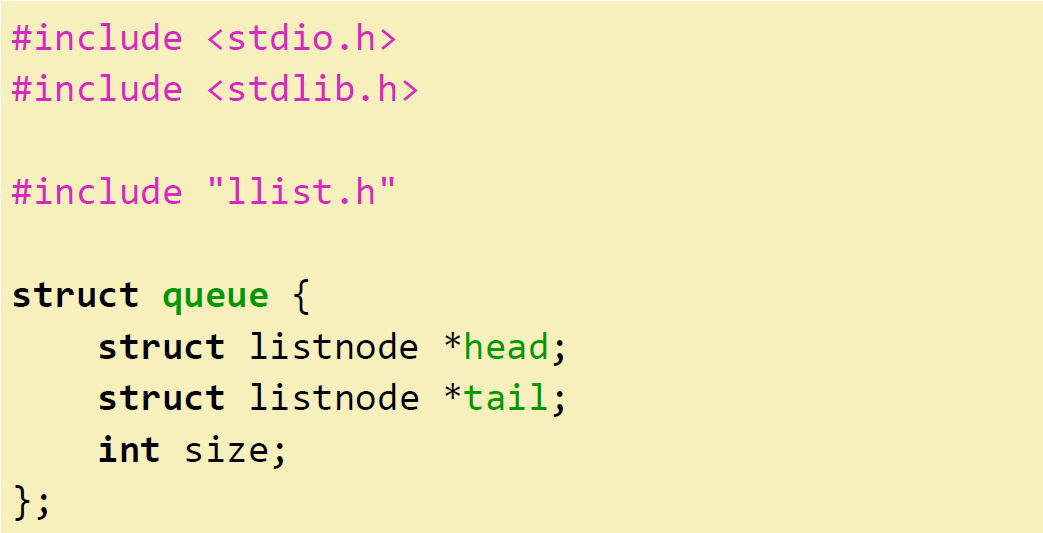
**1.На основе связных списков**Длина очереди ограничена лишь объемом доступной памяти

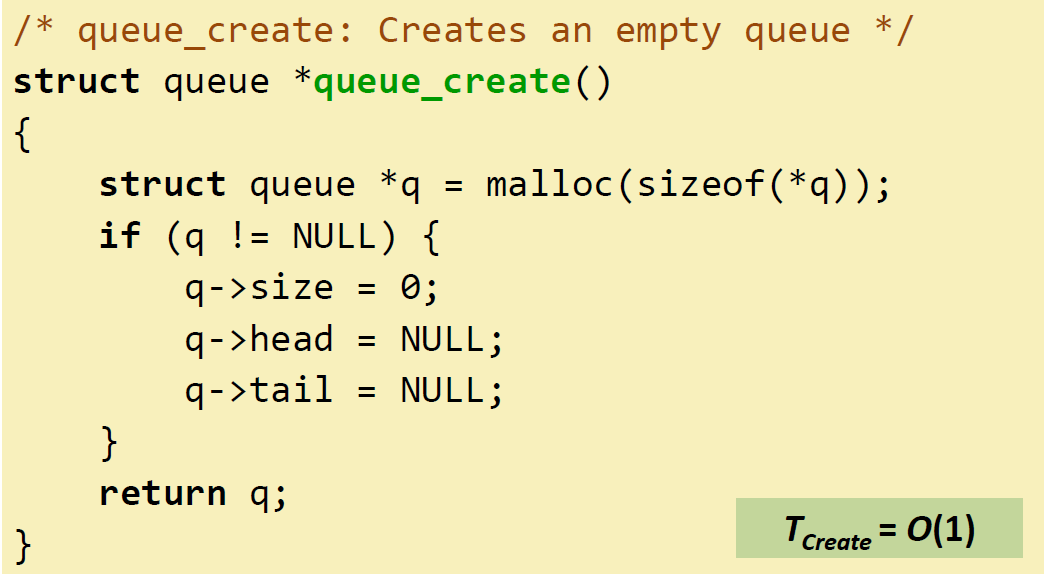
**2.На основе статических массивов**Длина очереди фиксирована (задана максимальная длина)

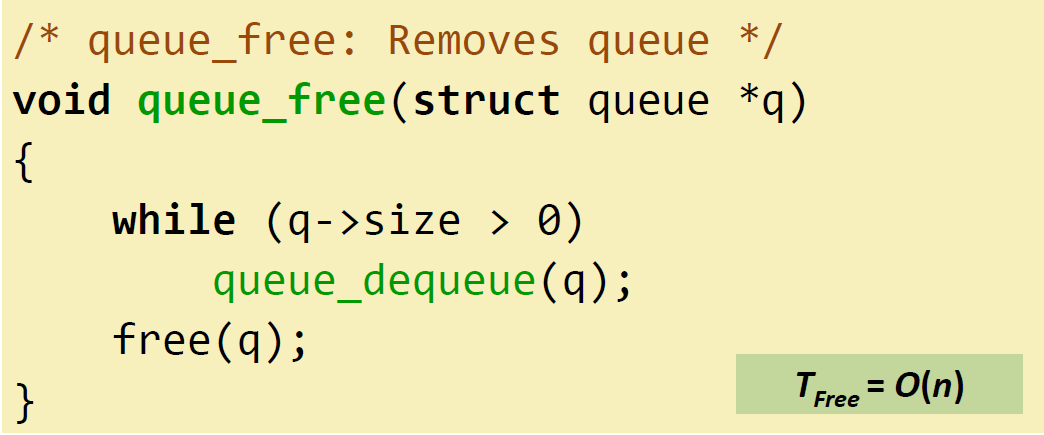


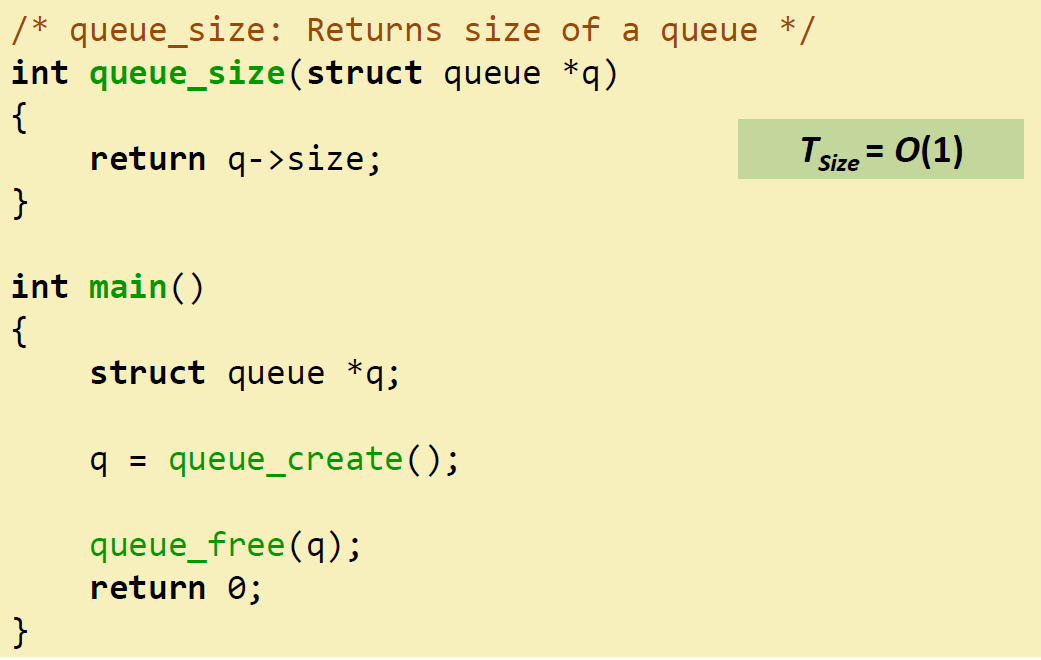
**Преимущества**: длина очереди ограничена лишь объемом доступной памяти

**Недостатки**(по сравнению с реализацией на основе массивов): работа с очередью немного медленнее, требуется больше памяти для хранения одного элемента

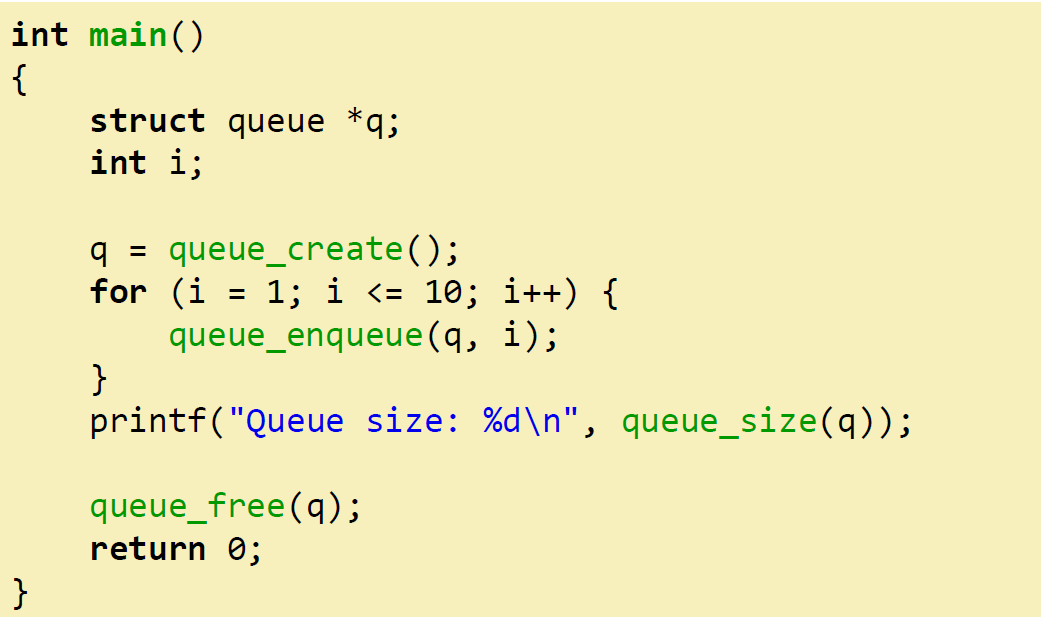


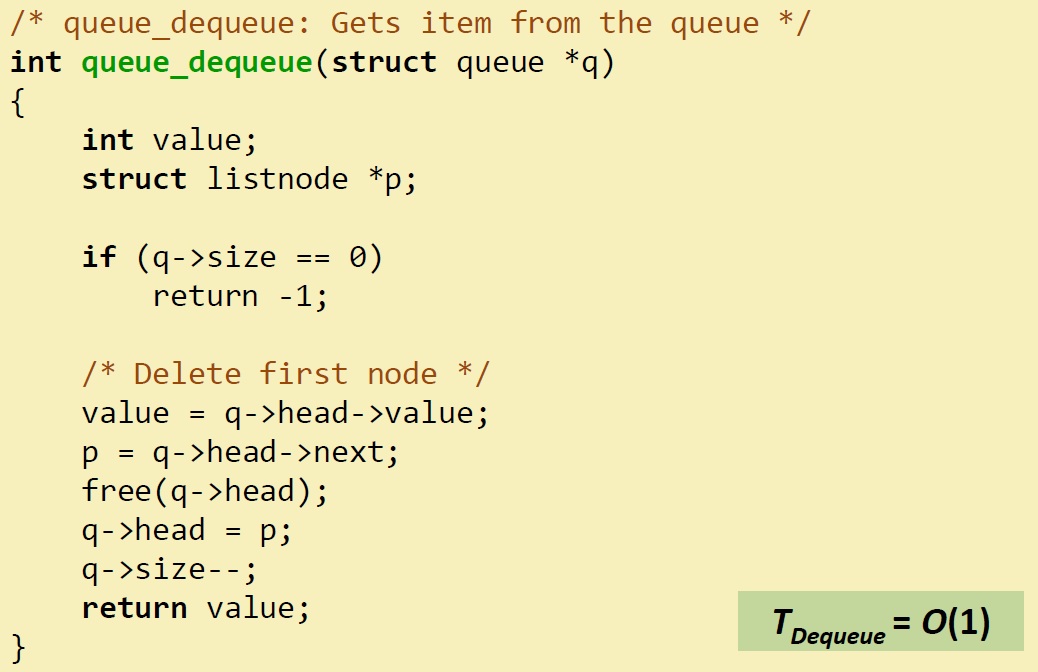


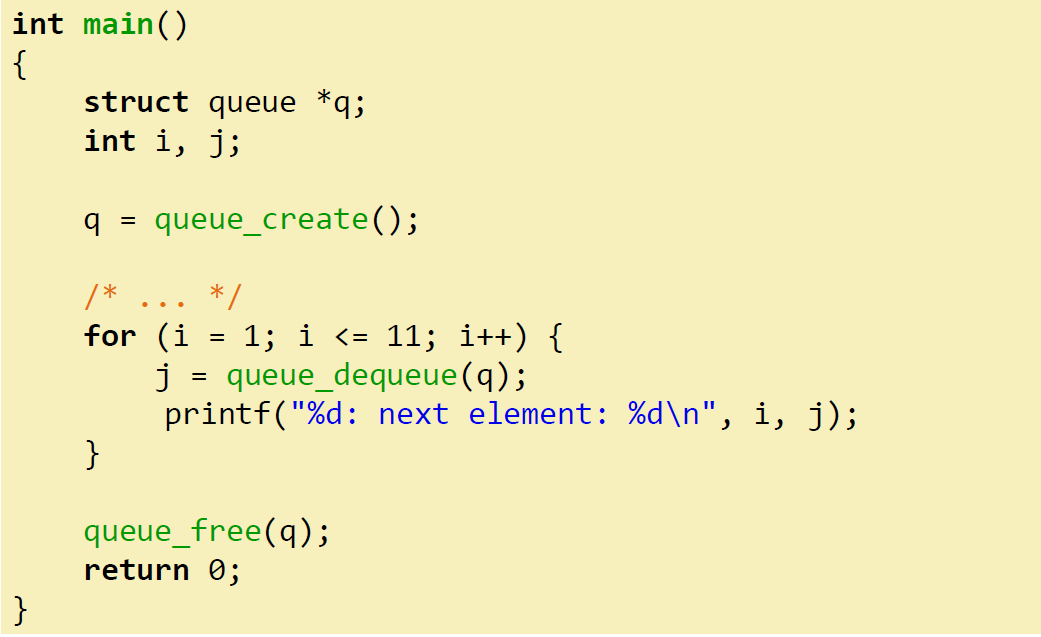


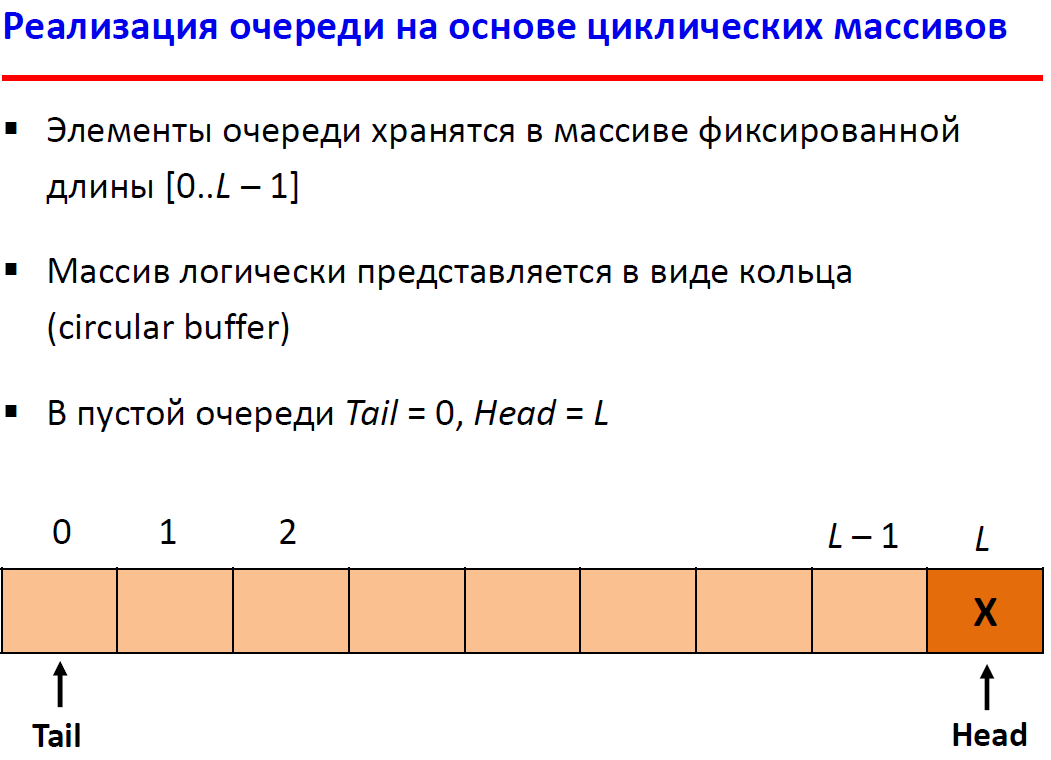












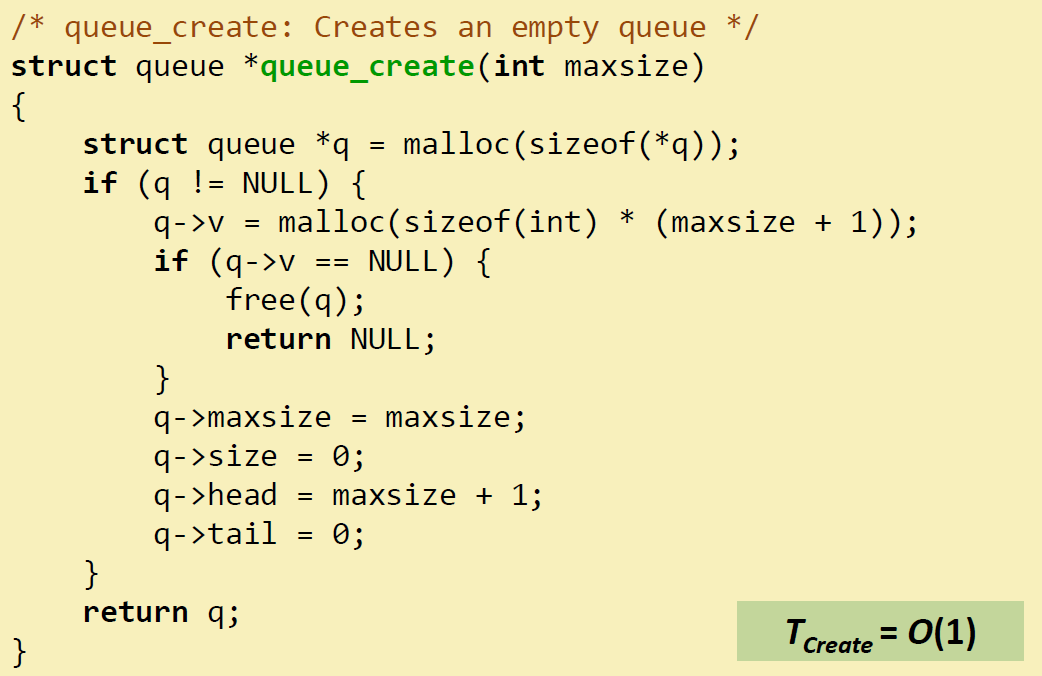
При **добавлении** элемента в очередь значение *Tail* циклически увеличивается на 1 (сдвигается на следующую свободную позицию)

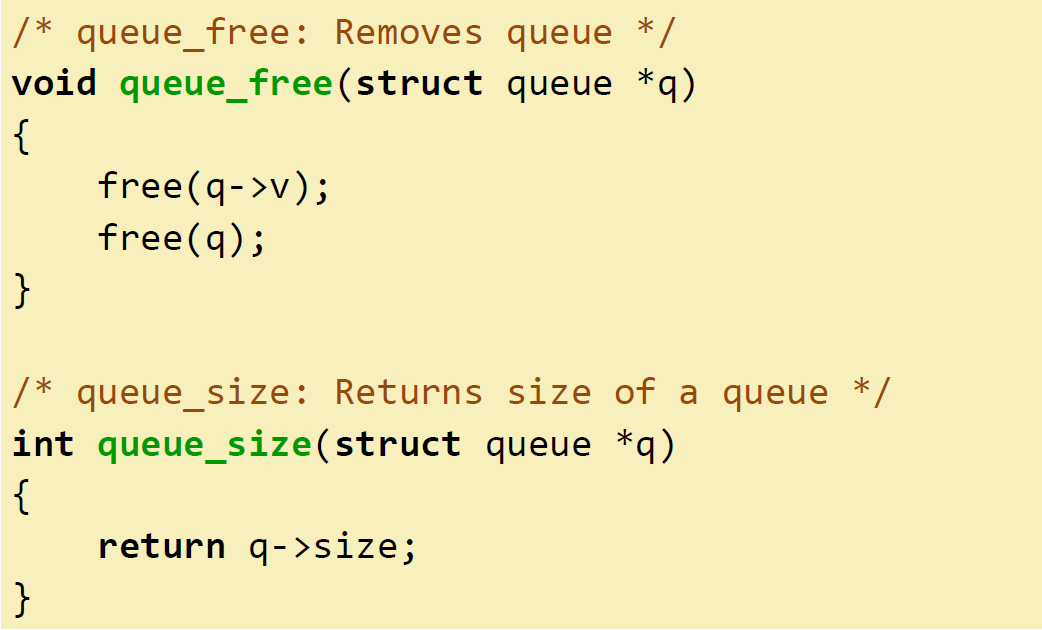
|  |
| --- |
| Если *Head*=*Tail*+1,то очередь переполнена! |

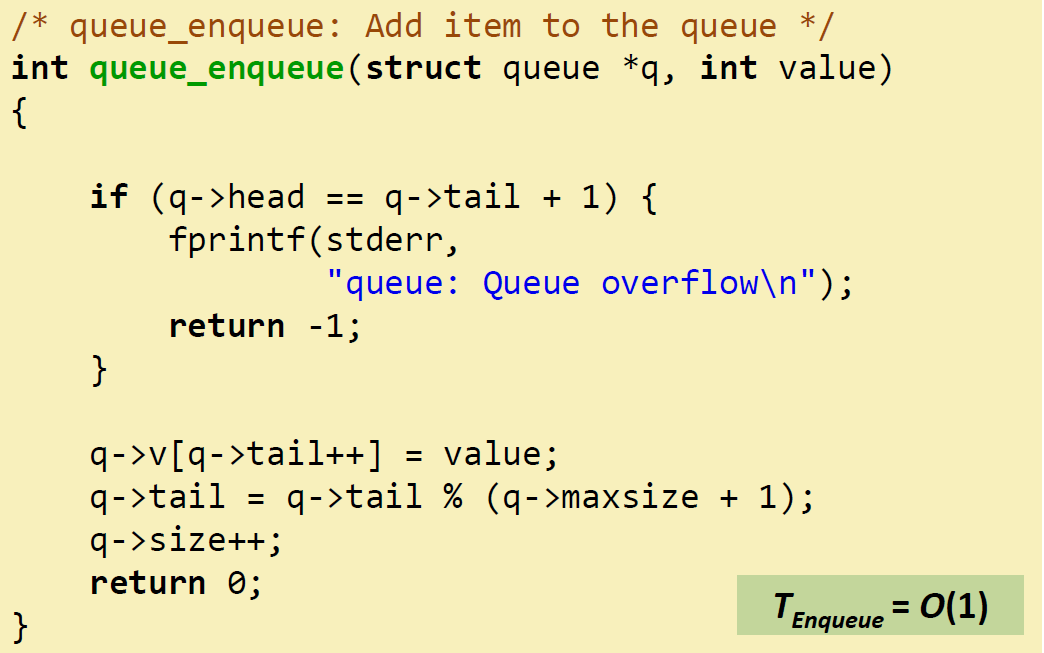
При **удалении** возвращается элемент с номером *Head* % *L*

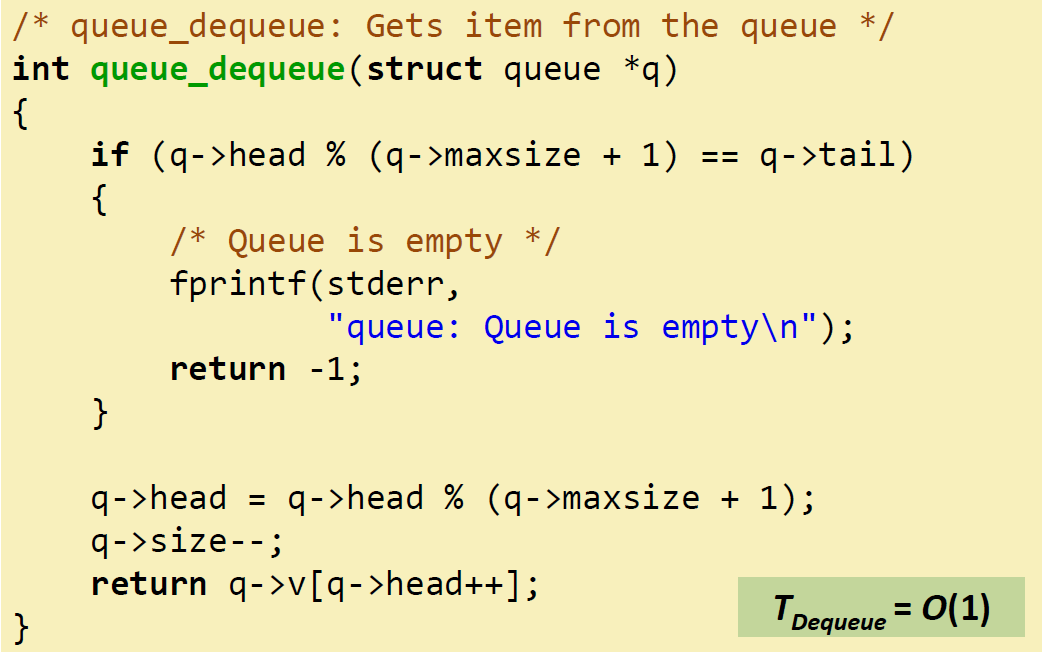
Значение *Head* циклически увеличивается на 1 (указывает на следующий элемент очереди)

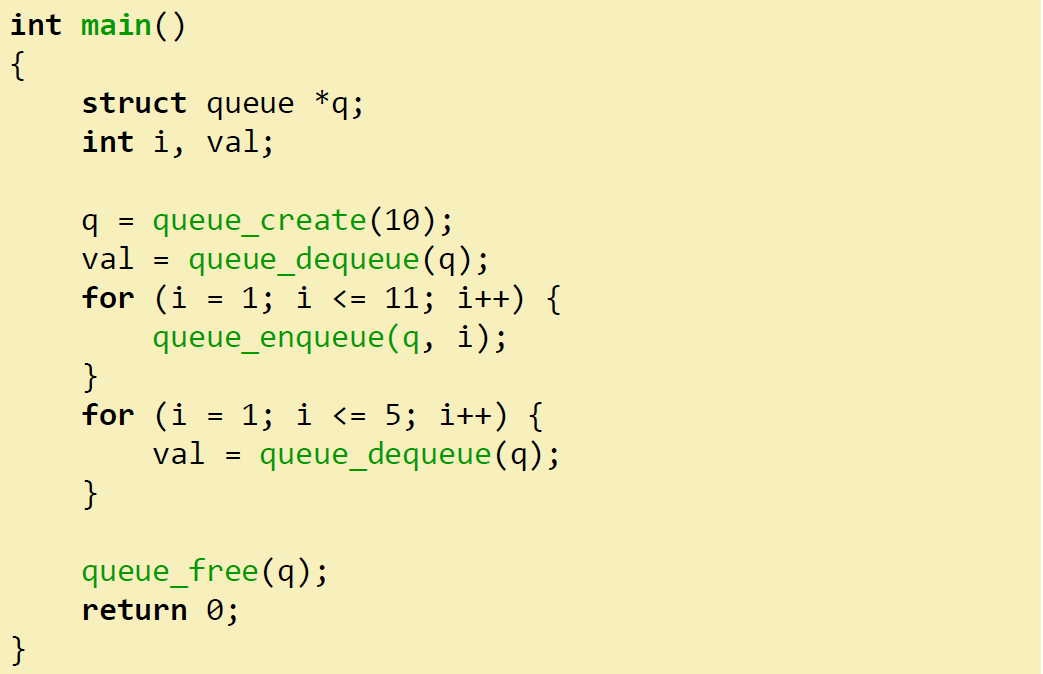








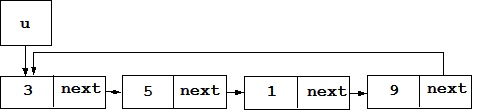




***8. Кольцевой список***

Кольцевой список — это список, у которого последний элемент связан с первым. Кольцевой список можно сделать как односвязным, так и двухсвязным. Рассмотрим вкратце односвязный кольцевой список.

Схема кольцевого списка представлена на рисунке ниже (используем те же данные, что и для ранее рассмотренного односвязного списка, т.е. список из чисел **3, 5, 1, 9**):

Используем те же структуры (**Data** и **List**), что применяли для односвязного списка. Точно так же определяем указатель на начало будущего списка:

**List \*u = NULL;**

и так же делаем начальное заполнение списка. Но в конце, после того, как для нашего примера мы занесли число **9** в список, требуется «замкнуть» список на начало:

**x->next = u;**

В итоге будет получен кольцевой список.

Операции для кольцевого списка можно выполнять те же, что и для обычного списка, но здесь будет присутствовать одна особенность: список замкнут, поэтому проверка того факта, что достигнут конец цикла, выполняется по другому. Покажем это на примере распечатки кольцевого цикла:

**void printC(List \*u)**

**{**

**if(u != NULL)**

**{**

**List \*p = u;**

**cout << "Kolcevoj Spisok:" << endl;**

**cout << p->d.a << endl;**

**p = p->next;**

**while(p != u)**

**{**

**cout << p->d.a << endl;**

**p = p->next;**

**}**

**}**

**else**

**cout << "Spisok pust." << endl;**

**}**

Возможный вызов функции:

**printC(u);**

Как видно из примера, после печати первого элемента, мы печатаем не до конца списка, а до нахождения начала.

Другие операции для кольцевого списка разработайте самостоятельно.

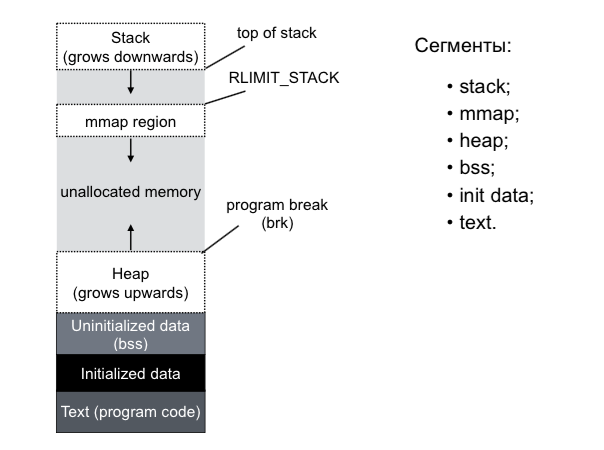
**9. ДВУНАПРАВЛЕННЫЙ СПИСОК**

**Двусвязный список (двунаправленный связный список)**

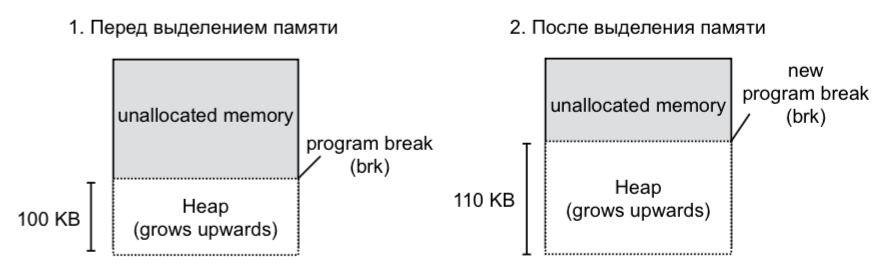
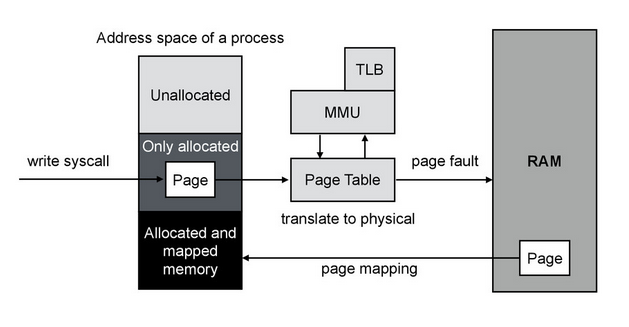
Здесь ссылки в каждом узле указывают на предыдущий и на последующий узел в списке. Как и односвязный список, двусвязный допускает только последовательный доступ к элементам, но при этом дает возможность перемещения в обе стороны. В этом списке проще производить удаление и перестановку элементов, так как легко доступны адреса тех элементов списка, указатели которых направлены на изменяемый элемент.

10 Распределение памяти в процессе

### Память процесса

Процесс состоит из следующих сегментов. У нас есть stack, который растет вниз; у него есть лимит дальше котрого он расти не может.   
  
  
  
Затем идет регион mmap: там находятся все отображенные на память файлы процесса, которые мы открыли или создали через системный вызов mmap(). Далее идет большое пространство невыделенной виртуальной памяти, которую мы можем использовать. Снизу вверх растет heap – это область анонимной памяти. Внизу идут области бинарника, который мы запускаем.  
  
Если мы говорим о памяти внутри процесса, то работать со страницами тоже неудобно: как правило, выделение памяти внутри процесса происходит блоками. Очень редко требуется выделить одну­-две странички, обычно нужно выделить сразу какой-­то промежуток страниц. Поэтому в Linux существует такое понятие, как область памяти (virtual memory area, VMA), которая описывает какое-­то пространство адресов внутри виртуального адресного пространства этого процесса. На каждую такую VMA есть свои права (чтения, записи, исполнения) и области видимости: она может быть приватная или общая (которая «шарится (share)» с другими процессами в системе).

### Выделение памяти

Выделение памяти можно поделить на четыре случая: есть выделение приватной памяти и памяти, которой можем с кем-­то поделиться (share); двумя другими категорями являются разделение на анонимную память и ту, у которая связана с файлом на диске. Самые частые функции выделения памяти – это malloc и free. Если мы говорим о glibc malloc(), то он выделяет анонимную память таким интересным способом: использует heap для аллокации маленьких объемов (менее 128 KБ) и mmap() для больших объемов. Такое выделение необходимо для того, чтобы память расходовалась оптимальнее и её можно было запросто отдавать в систему.  Если в heap не хватает памяти для выделения, вызывается системный вызов brk(), который расширяет границы heap. Системный вызов mmap() занимается тем, что отображает содержимое файла на адресное пространство. munmap() в свою очередь освобождает отображение. У mmap() есть флаги, которые регулируют видимость изменений и уровень доступа.   
  
  
  
На самом деле, Linux не выделяет всю запрошенную память сразу. Процесс выделения памяти — Demand Paging — начинается с того, что мы запрашиваем у ядра системы страничку памяти, и она попадает в область Only Allocated. Ядро отвечает процессу: вот твоя страница памяти, ты можешь её использовать. И больше ничего происходит. Никакой физической аллокации не происходит. А произойдет она только в том случае, если мы попробуем в эту страницу произвести запись. В этот момент пойдёт обращение в Page Table – эта структура транслирует виртуальные адреса процесса в физические адреса оперативной памяти. При этом будут задействованы также два блока: MMU и TLB, как видно из рисунка. Они позволяют ускорять выделение и служат для трансляции виртуальных адресов в физические.  
  
  
  
После того, как мы понимаем, что этой странице в Page Table ничего не соответствует, то есть нет связи с физической памятью, мы получаем Page Fault – в данном случае минорный (minor), так как отсутствует обращение в диск. После этого процесса система может производить запись в выделенную страницу памяти. Для процесса все это происходит прозрачно. А мы можем наблюдать увеличение счетчика минорных Page Fault для процесса на одну единицу. Также бывает мажорный Page Fault – в случае, когда происходит обращение в диск за содержимым страницы (в случае mmpa()).  
  
Один из трюков в работе с памятью в Linux – Copy On Write – позволяет делать очень быстрые порождения процессов (fork).  
  
**Работа с файлами и с памятью**

Подсистема памяти и подсистема работы с файлами тесно связаны. Так как работа с диском напрямую очень медленна, ядро использует в качестве прослойки оперативную память.  
  
malloc() использует больше памяти: происходит копирование в user space. Также потребляется больше CPU, и мы получаем больше переключений контекста, чем если бы мы работали с файлом через mmap().  
  
Какие выводы можно сделать? Мы можем работать с файлами, как с памятью. У нас есть lazy lоading, то есть мы можем замапить очень­-очень большой файл, и он будет подгружаться в память процесса через Page Cache только по мере надобности. Всё также происходит быстрее, потому что мы используем меньше системных вызовов и, в конце концов, это экономит память. Ещё стоит отметить, что при завершении программы память никуда не девается и остается в Page Cache.  
  
В начале было сказано, что вся запись и чтение идут через Page Cache, но иногда по какой­-то причине, есть необходимость в отходе от такого поведения. Некоторые программные продукты работают таким способом, например MySQL с InnoDB.  
  
Подсказать ядру, что в ближайшее время мы не будем работать с этим файлом, и заставить выгрузить страницы файла из Page Cache можно с помощью специальных системных вызовов:

* posix\_fadvide();
* madvise();
* mincore().

Утилита vmtouch также может вынести страницы файла из Page Cache – ключ “­e”.

**11. Перечисляемый тип enum.**

В си выделен отдельный тип перечисление (enum), задающий набор всех возможных целочисленных значений переменной этого типа. Синтаксис перечисления:

|  |
| --- |
| enum <имя> {<имя поля 1>, <имя поля 2>, ... , <имя поля N>}; |

**12. Бинарное дерево поиска. Сложности работы. Плюсы и минусы**

*Ответ на вопрос разбит на две части - непосредственно сам ответ и материал для понимания. Начнём с первого.*

Дерево – структура данных, представляющая собой древовидную структуру в виде набора связанных узлов.

*Бинарное (двоичное) дерево поиска* – это бинарное дерево, для которого выполняются следующие дополнительные условия (свойства дерева поиска):

1. оба поддерева – левое и правое, являются двоичными деревьями поиска;
2. у всех узлов левого поддерева произвольного узла X значения ключей данных меньше, чем значение ключа данных самого узла X;
3. у всех узлов правого поддерева произвольного узла X значения ключей данных не меньше, чем значение ключа данных узла X.

Данные в каждом узле должны обладать ключами, на которых определена операция сравнения меньше. Как правило, информация, представляющая каждый узел, является записью, а не единственным полем данных.

Плюсы:

+возможность быстро структурировать данные;

+присутствует классификация (часть элементов в левом поддереве, часть – в правом);

Минусы:

-сложное удаление;

-необходимость в балансировке (если дерево несбалансировано, оно не выгодно)

**Инициализация**:

node \*create(node \*root, int key)

{

node \*tmp = malloc(sizeof(node)); // Выделение памяти под корень дерева

tmp -> key = key; // Присваивание значения ключу

tmp -> parent = NULL; // Присваивание указателю на родителя значения NULL

tmp -> left = tmp -> right = NULL; // Присваивание указателю на левое и правое поддерево значения NULL

root = tmp;

return root;

}

**Добавление узлов в дерево**

struct tnode \* addnode(int x, tnode \*tree) {

if (tree == NULL) { // Если дерева нет, то формируем корень

tree =new tnode; // память под узел

tree->field = x; // поле данных

tree->left = NULL;

tree->right = NULL; // ветви инициализируем пустотой

}else if (x < tree->field) // условие добавление левого потомка

tree->left = addnode(x,tree->left);

else // условие добавление правого потомка

tree->right = addnode(x,tree->right);

return(tree);

}

**Удаление поддерева:**

void freemem(tnode \*tree) {

if(tree!=NULL) {

freemem(tree->left);

freemem(tree->right);

delete tree;

}

}

13 – это тоже самое, что 14

14. Критерии оценки сложности и эффективности алгоритмов

Сложность алгоритмов обычно оценивают по времени выполнения или по используемой памяти. Формально O(f(n)) означает, что время работы алгоритма (или объём занимаемой памяти) растёт в зависимости от объёма входных данных не быстрее, чем некоторая константа, умноженная на f(n).

O(n) — линейная сложность

Такой сложностью обладает, например, алгоритм поиска наибольшего элемента в не отсортированном массиве. Нам придётся пройтись по всем n элементам массива, чтобы понять, какой из них максимальный.

public long GetSum(int[] items)

{

long sum = 0;

foreach (int i in items)

{

sum += i;

}

return sum;

}

O(log n) — логарифмическая сложность

Простейший пример — бинарный поиск. Если массив отсортирован, мы можем проверить, есть ли в нём какое-то конкретное значение, методом деления пополам. Проверим средний элемент, если он больше искомого, то отбросим вторую половину массива — там его точно нет. Если же меньше, то наоборот — отбросим начальную половину. И так будем продолжать делить пополам, в итоге проверим log n элементов.

O(n2) — квадратичная сложность

Такую сложность имеет, например, алгоритм сортировки вставками. В канонической реализации он представляет из себя два вложенных цикла: один, чтобы проходить по всему массиву, а второй, чтобы находить место очередному элементу в уже отсортированной части. Таким образом, количество операций будет зависеть от размера массива как n \* n, т. е. n2.

Также случается, что время работы алгоритма вообще не зависит от размера входных данных. Тогда сложность обозначают как O(1).

public int GetCount(int[] items){return items.Length;}

15. **Хэш-табли́ца** — это [структура данных](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), реализующая интерфейс [ассоциативного массива](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D1%81%D0%BE%D1%86%D0%B8%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2), а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.

Для каждого добавляемого значения вычисляется его хэш, в ячейку массива с номером, равным хэшу, кладётся информация.

**Хеширование** — преобразование [массива](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D1%81%D0%B8%D0%B2_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) входных данных произвольной длины в (выходную) [битовую](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D1%82) строку установленной длины, выполняемое [определённым алгоритмом](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D1%82%D0%B5%D1%80%D0%BC%D0%B8%D0%BD%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC). Функция, воплощающая алгоритм и выполняющая преобразование, называется «*хеш-функцией*» или «*функцией свёртки*». Исходные данные называются входным массивом, «ключом» или «*сообщением*». Результат преобразования (выходные данные) называется «[*хешем*](https://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88-%D1%81%D1%83%D0%BC%D0%BC%D0%B0)», «*хеш-кодом*», «*хеш-суммой*», «сводкой [сообщения](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%BE%D0%B1%D1%89%D0%B5%D0%BD%D0%B8%D0%B5)».

Хэш-таблица отличается тем, что позволяет обработать информацию в ней за гарантированное время, так как всё, что необходимо – вычислить хэш и обратиться по нему к нужной ячейке памяти, перебор всех записей исключён.

Узким местом таких форм организации данных являются:  
1.Масштабирование(расширение таблицы), так как требует смены значения делителя хэша и пересчёта всех ключей

2.Коллизия –совпадение ключей для определённых входных данных. Решается созданием связного списка внутри таблицы.

16. . Алгоритм qsort, сложность, плюсы\минусы.

Итак, быстрая сортировка, или, по названию функции в Си, Qsort — это алгоритм сортировки, сложность которого в среднем составляет O(n log(n)). Суть его предельно проста: выбирается так называемый опорный элемент, и массив делится на 3 подмассива: меньших опорного, равных опорному и больших опорного. Потом этот алгоритм применяется рекурсивно к подмассивам.

# Алгоритм

1. Выбираем опорный элемент
2. Разбиваем массив на 3 части  
   * Создаём переменные l и r — индексы соответственно начала и конца рассматриваемого подмассива
   * Увеличиваем l, пока l-й элемент меньше опорного
   * Уменьшаем r, пока r-й элемент больше опорного
   * Если l всё ещё меньше r, то меняем l-й и r-й элементы местами, инкрементируем l и декрементируем r
   * Если l вдруг становится больше r, то прерываем цикл
3. Повторяем рекурсивно, пока не дойдём до массива из 1 элемента

Что ж, выглядит не так уж сложно. Реализуем на Си? Нет проблем!

[void qsort (int b, int e)  
{  
  int l = b, r = e;  
  int piv = arr[(l + r) / 2]; // Опорным элементом для примера возьмём средний  
  while (l <= r)  
  {  
    while (arr[l] < piv)  
      l++;  
    while (arr[r] > piv)  
      r--;  
    if (l <= r)  
      swap (arr[l++], arr[r--]);  
  }  
  if (b < r)  
    qsort (b, r);  
  if (e > l)  
    qsort (l, e);  
}    /\* ----- end of function qsort ----- \*/  
  
// qsort (0, n-1);  
\* This source code was highlighted with Source Code Highlighter](http://virtser.net/blog/post/source-code-highlighter.aspx).

Эта реализация имеет ряд недостатков, таких как возможное переполнение стека из-за большого количества вложенной рекурсии и то, что опорным элементом всегда берётся средний. Для примера это, может, и нормально, но при решении, например, олимпиадных задач, хитрое жюри может специально подобрать такие тесты, чтобы на них это решение работало слишком долго и не проходило в лимит. В принципе, в качестве опорного элемента можно брать любой, но лучше, чтобы он был максимально приближен к медиане, поэтому можно выбрать его случайно или взять средний по значению из первого, среднего и последнего. Зависимость быстродействия от опорного элемента — один из недостатков алгоритма, ничего с этим не поделать, но сильная деградация производительности происходит редко, обычно если сортируется специально подобранный набор чисел. Если всё-таки нужна сортировка, работающая гарантированно быстро, можно использовать, например, пирамидальную сортировку, всегда работающую строго за O(n log n). Обычно Qsort всё же выигрывает в производительности перед другими сортировками, не требует много дополнительной памяти и достаточно прост в реализации, поэтому пользуется заслуженной популярностью.

Вопрос № 17

В программировании **заголовочный файл** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *header file*) или **подключаемый файл** — файл, содержимое которого автоматически добавляется [препроцессором](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80) в [исходный текст](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D1%82%D0%B5%D0%BA%D1%81%D1%82) в том месте, где располагается некоторая [директива](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D1%80%D0%B5%D0%BA%D1%82%D0%B8%D0%B2%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) (#include <file.h> в Си).

В языках программирования [Си](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) и [C++](https://ru.wikipedia.org/wiki/C%2B%2B) заголовочные файлы — основной способ подключить к программе [типы данных](https://ru.wikipedia.org/wiki/%D0%A2%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), структуры, [прототипы функций](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%82%D0%BE%D1%82%D0%B8%D0%BF_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8), [перечислимые типы](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D0%BC%D1%8B%D0%B9_%D1%82%D0%B8%D0%BF) и [макросы](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D0%BA%D1%80%D0%BE%D1%81), используемые в другом модуле. [По умолчанию](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE_%D1%83%D0%BC%D0%BE%D0%BB%D1%87%D0%B0%D0%BD%D0%B8%D1%8E) используется [расширение](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D1%81%D1%88%D0%B8%D1%80%D0%B5%D0%BD%D0%B8%D0%B5_%D0%B8%D0%BC%D0%B5%D0%BD%D0%B8_%D1%84%D0%B0%D0%B9%D0%BB%D0%B0) *.h*; иногда для заголовочных файлов языка [C++](https://ru.wikipedia.org/wiki/C%2B%2B) используют расширение *.hpp*.

Чтобы избежать повторного включения одного и того же кода, используются директивы #ifndef, #define, #endif.

Заголовочный файл в общем случае может содержать любые конструкции [языка программирования](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), но на практике исполняемый код в заголовочные файлы не помещают. Например, [идентификаторы](https://ru.wikipedia.org/wiki/%D0%98%D0%B4%D0%B5%D0%BD%D1%82%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%82%D0%BE%D1%80), которые должны быть объявлены более чем в одном файле, удобно описать в заголовочном файле, а затем его подключать по мере надобности. Подобным же образом работает [модульность](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D1%83%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D1%8C_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) и в большинстве [ассемблеров](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D1%81%D0%B5%D0%BC%D0%B1%D0%BB%D0%B5%D1%80).

По сложившейся традиции, в заголовочных файлах объявляют функции [стандартной библиотеки](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D0%BD%D0%B0%D1%8F_%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0) [Си](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D0%BD%D0%B0%D1%8F_%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_%D1%8F%D0%B7%D1%8B%D0%BA%D0%B0_%D0%A1%D0%B8) и [Си++](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D0%BD%D0%B0%D1%8F_%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_C%2B%2B).

В современных языках программирования программы составляются из [модулей](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D1%83%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), компилируемых по отдельности. В связи с этим возникает вопрос: как указать, что [подпрограмма](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B4%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0) или [переменная](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) X определена в модуле Y? Для этого существует несколько решений, в [Си](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) применено такое.

В одной из [единиц компиляции](https://ru.wikipedia.org/wiki/%D0%95%D0%B4%D0%B8%D0%BD%D0%B8%D1%86%D0%B0_%D0%BA%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%86%D0%B8%D0%B8) (то есть с-файле) описывается функция, например:

int add(int a, int b)

{

**return** a + b;

}

Чтобы на неё можно было ссылаться из других единиц компиляции, требуется объявить её при помощи [прототипа функции](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%82%D0%BE%D1%82%D0%B8%D0%BF_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8), то есть:

int add(int, int);

int triple(int x)

{

**return** add(x, add(x, x));

}

Тем не менее, такое объявление требует, чтобы программист обеспечил объявление функции для add в двух местах — в файле, содержащем её реализацию, и в файле, в котором она используется. В случае изменения определения функции программист должен не забыть обновить все прототипы, использованные в программе.

Заголовочный файл является одним из решений этой проблемы. В заголовочном файле [модуля](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D1%83%D0%BB%D1%8C_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) объявляется каждая [функция](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), [объект](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) и [тип данных](https://ru.wikipedia.org/wiki/%D0%A2%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), являющиеся частью *интерфейса вызова* модуля — например, в этом случае заголовочный файл может содержать только объявление функции add. Каждый исходный файл, ссылающийся на функцию add, должен использовать директиву #include для подключения заголовочного файла:

*/\* File triple.c \*/*

#include *"add.h"*

int triple(int x)

{

**return** add(x, add(x, x));

}

**18. Инструменты сборки проектов make and makefile**

***make*** — это утилита для автоматической сборки программ. Позволяет отслеживать изменения в исходном коде программы и компилировать не весь проект целиком а только те файлы которые изменились или те которые зависят от внесенных изменений. При больших проектах это дает существенную экономию времени

По умолчанию правила сборки считываются из файла с именем Makefile.

Структуру Makefile можно представить так:

ЦЕЛЬ: ЗАВИСИМОСТЬ  
[TAB]ДЕЙСТВИЕ

Но обычно используются более сложные правила, например:

ЦЕЛЬ: ЦЕЛЬ1 ЦЕЛЬ2  
[TAB]ДЕЙСТВИЕ  
  
ЦЕЛЬ1: ЗАВИСИМОСТЬ1  
[TAB]ДЕЙСТВИЕ1  
  
ЦЕЛЬ2: ЗАВИСИМОСТЬ2  
[TAB]ДЕЙСТВИЕ2

ЦЕЛЬ — это то что мы получаем в результате ДЕЙСТВИЯ. Это может быть файл, директория или просто абстрактная ЦЕЛЬ не имеющая связи с каким-либо объектом на жестком диске. После имени цели ставится двоеточие. При запуске команды make без параметров выполнится первое найденное правило. Что бы выполнить другое правило надо указать его команде make

make ЦЕЛЬ2

ЗАВИСИМОСТЬ — это то, от чего зависит наша ЦЕЛЬ. Это могут быть файлы, каталоги или другие ЦЕЛИ. Make сравнивает дату и время изменения ЦЕЛИ и объектов о которых зависит цель. Если объекты от которых зависит цель были изменены позже чем создана цель, то будет выполнено ДЕЙСТВИЕ. ДЕЙСТВИЕ так же выполняется если ЦЕЛЬ не является именем файла или директории.

ДЕЙСТВИЕ — это набор команд которые надо выполнить. Перед командами должен быть введен символ табуляции. Если вместо символа табуляции будут введены пробелы то при компиляции будет выведено сообщение об ошибке:

Makefile:13: \*\*\* пропущен разделитель. Останов.

или

Makefile:13: \*\*\* missing separator. Stop.

Пример Makefile:

|  |
| --- |
| test.elf: test.c  gcc test.c -o test.elf |

Пример с абстрактной целью:

|  |
| --- |
| all: test.elf   test.elf: test1.o test2.o  gcc -o test.elf test1.o test2.o   test1.o test1.c  gcc -c test1.c -o test1.o   test2.o test2.c  gcc -c test2.c -o test2.o |

Рассмотри последний пример:

Первым выполняется all т.к. находится в начале Makefile. all зависит от test.elf и файла или директории с именем all не существует, поэтому всегда будет происходить проверка цели с именем test.elf.

test.elf зависит от test1.o и test2.o, по этому сначала будет проверена цель test1.o затем test2.o

При проверке цели test1.o сравниваются дата и время изменения файла test1.o и test1.c. Если файл test1.o не существует или файл test1.c был изменены позднее чем test1.o то будет выполнена команда gcc -с test1.c -o test1.o.

Аналогично будет проверена цель test2.o.

После этого сравниваются дата и время изменения файла test.elf и файлов test1.o test2.o. Если test1.o или test2.o новее то будет выполнена команда gcc -o test.elf test1.o test2.o

Таким образом отслеживаются изменения в файлах test1.с и test2.c.

19. Указатели на Функции в Си

Указатель на функцию содержит адрес первого байта в памяти, по которому располагается выполняемый код функции.Самым распространенным указателем на функцию является ее имя. С помощью имени функции мы можем вызывать ее и получать результат ее работы.Но также указатель на функцию мы можем определять в виде отдельной переменной с помощью следующего синтаксиса:

1тип (\*имя\_указателя) (параметры);

Здесь тип представляет тип возвращаемого функцией значения.имя\_указателя представляет произвольно выбранный идентификатор в соответствии с правилами о наименовании переменных.И параметры определяют тип и название параметров через запятую при их наличии.Например, определим указатель на функцию:

1void (\*message) (void);>

Здесь определен указатель, который имеет имя message. Он может указывать на функции без параметров, которые возвращают тип void (то есть ничего не возвращают).

Указателю на функцию можно присвоить функцию, которая соответствует указателю по возвращаемому типу и спецификации параметров:

1message=hello;То есть в данном случае указатель message теперь хранит адрес функции hello. И посредством обращения к указателю мы можем вызвать эту функцию:

1message();Впоследствии мы можем присвоит указателю адрес другой функции, как в данном случае.

20. Указатели на функции, обратные вызовы (callback)

 Как Выполнить Обратный Вызов в C?

Я только беру объявление функции qsort, которое читается следующим образом: void qsort( void\* field, size\_t nElements, size\_t sizeOfAnElement, int(\_USERENTRY \*cmpFunc)(**const** void\*, **const** void\*)); **field** указывает на первый элемент области, которая должна быть отсортирована, **nElements** - число элементов в области, **sizeOfAnElement** - размер одного элемента в байтах и **cmpFunc** - указатель на функцию сравнения. Эта функция сравнения принимает два указателя на void и озвращает int. Синтаксис использования указателя на функцию в качестве параметра выглядит немного странно. Посмотрите только, как объявить указатель на функцию, и вы увидите, что в действительности это одно и то же. Обратный вызов выполняется точно так же, как и обычная функция: только вы используете имя указателя на функцию вместо имени функции. Это показано ниже. Замечание: Все аргументы вызова, кроме указателя на функцию, были опущены, чтобы сконцентрироваться на важных вещах.

void qsort( ... , int(\_USERENTRY \*cmpFunc)(const void\*, const void\*))

{

/\* алгоритм сортировки - замечание: item1 и item2 - имеют тип void\* \*/

int bigger=cmpFunc(item1, item2); // проиводится обратный вызов

/\* использование результата \*/

}

3.3  Код примера использования qsort

В следующем приметре сортируется массив элементов типа float.

//-----------------------------------------------------------------------------------------

// 3.3 Как выполнить обратный вызов в C с использованием функции сортировки qsort

#include <stdlib.h> // необходимо для:qsort

#include <time.h> // randomize

#include <stdio.h> // printf

// функция сравнения для алгоритма сортировки

// два элемента передаются как указатели на void, преобразуются и сравниваются

int CmpFunc(const void\* \_a, const void\* \_b)

{

// вы должны выполнить явное приведение к правильному типу

const float\* a = (const float\*) \_a;

const float\* b = (const float\*) \_b;

if(\*a > \*b) return 1; // первый элемент больше, чем второй -> return 1

else

if(\*a == \*b) return 0; // равны -> return 0

else return -1; // второй элемент больше, чем первый -> return -1

}

// пример использования qsort()

void QSortExample()

{

float field[100];

::randomize(); // инициализация генератора случайных чисел

for(int c=0;c<100;c++) // заполнение массива случайными числами

field[c]=random(99);

// сортировка, используя qsort()

qsort((void\*) field, /\*количество элементов\*/ 100, /\*размер элемента\*/ sizeof(field[0]),

/\*функция сравнения\*/ CmpFunc);

// отображение первых десяти элементов отсортированного массива

printf("Первые десять элементов отсортированного массива ...\n");

for(int c=0;c<10;c++)

printf("элемент #%d содержит %.0f\n", c+1, field[c]);

printf("\n");

}

/\* Мастер-класс по созданию функции обратного вызова. \*/

#include <stdio.h>

#include <stdlib.h>

int values[] = { 40, 10, 100, 90, 20, 25 };

int compare( const void \*a, const void \*b )

{

return ( \*(int \*)a - \*(int \*)b );

}

int main ()

{

int n;

qsort (values, 6, sizeof(int), compare);

for (n=0; n<6; n++)

printf ("%d ",values[n]);

return 0;

}

21.

## Глава 3. БИБЛИОТЕКИ

[3.1. Введение в библиотеки](http://www.opennet.ru/docs/RUS/zlp/003.html#1)  
[3.2. Пример статической библиотеки](http://www.opennet.ru/docs/RUS/zlp/003.html#2)  
[3.3. Пример совместно используемой библиотеки](http://www.opennet.ru/docs/RUS/zlp/003.html#3)

### 3.1. Введение в библиотеки

Как уже неоднократно упоминалось в предыдущей главе, *библиотека* - это набор скомпонованных особым образом объектных файлов. Библиотеки подключаются к основной программе во время линковки. По способу компоновки библиотеки подразделяют на *архивы* (статические библиотеки, static libraries) и *совместно используемые* (динамические библиотеки, shared libraries). В Linux, кроме того, есть механизмы *динамической подгрузки* библиотек. Суть динамической подгрузки состоит в том, что запущенная программа может по собственному усмотрению подключить к себе какую-либо библиотеку. Благодаря этой возможности создаются программы с *подключаемыми плагинами*, такие как XMMS. В этой главе мы не будем рассматривать динамическую подгрузку, а остановимся на классическом использовании статических и динамических библиотек.

С точки зрения модели КИС, библиотека - это сервер. Библиотеки несут в себе одну важную мысль: возможность использовать одни и те же механизмы в разных программах. В Linux библиотеки используются повсеместно, поскольку это очень удобный способ "не изобретать велосипеды". Даже ядро Linux в каком-то смысле представляет собой библиотеку механизмов, называемых *системными вызовами*.

Статическая библиотека - это просто архив объектных файлов, который подключается к программе во время линковки. Эффект такой же, как если бы вы подключали каждый из файлов отдельно.

В отличие от статических библиотек, код совместно используемых (динамических) библиотек не включается в бинарник. Вместо этого в бинарник включается только *ссылка* на библиотеку.

Рассмотрим преимущества и недостатки статических и совместно используемых библиотек. Статические библиотеки делают программу более автономной: программа, скомпонованная со статической библиотекой может запускаться на любом компьютере, не требуя наличия этой библиотеки (она уже "внутри" бинарника). Программа, скомпонованная с динамической библиотекой, требует наличия этой библиотеки на том компьютере, где она запускается, поскольку в бинарнике не код, а ссылка на код библиотеки. Не смотря на такую зависимость, динамические библиотеки обладают двумя существенными преимуществами. Во-первых, бинарник, скомпонованный с совместно используемой библиотекой меньше размером, чем такой же бинарник, с подключенной к нему статической библиотекой (статически скомпонованный бинарник). Во-вторых, любая модернизация динамической библиотеки, отражается на всех программах, использующих ее. Таким образом, если некоторую библиотеку foo используют 10 программ, то исправление какой-нибудь ошибки в foo или любое другое улучшение библиотеки автоматически улучшает все программы, которые используют эту библиотеку. Именно поэтому динамические библиотеки называют совместно используемыми. Чтобы применить изменения, внесенные в статическую библиотеку, нужно пересобрать все 10 программ.

В Linux статические библиотеки обычно имеют расширение **.a** (Archive), а совместно используемые библиотеки имеют расширение **.so** (Shared Object). Хранятся библиотеки, как правило, в каталогах /lib и /usr/lib. В случае иного расположения (относится только к совместно используемым библиотекам), приходится немного "подшаманить", чтобы программа запустилась.

### 3.2. Пример статической библиотеки

Теперь давайте создадим свою собственную библиотеку, располагающую двумя функциями: h\_world() и g\_world(), которые выводят на экран "Hello World" и "Goodbye World" соответственно. Начнем со статической библиотеки.

Начнем с интерфейса. Создадим файл world.h:

**/\* world.h \*/**

**void h\_world (void);**

**void g\_world (void);**

Здесь просто объявлены функции, которые будут использоваться.

Теперь надо реализовать серверы. Создадим файл h\_world.c:

**/\* h\_world.c \*/**

**#include <stdio.h>**

**#include "world.h"**

**void h\_world (void)**

**{**

**printf ("Hello World\n");**

**}**

Теперь создадим файл g\_world.c, содержащий реализацию функции g\_world():

**/\* g\_world.c \*/**

**#include <stdio.h>**

**#include "world.h"**

**void g\_world (void)**

**{**

**printf ("Goodbye World\n");**

**}**

Можно было бы с таким же успехом уместить обе функции в одном файле (hello.c, например), однако для наглядности мы разнесли код на два файла.

Теперь создадим файл main.c. Это клиент, который будет пользоваться услугами сервера:

**/\* main.c \*/**

**#include "world.h"**

**int main (void)**

**{**

**h\_world ();**

**g\_world ();**

**}**

Теперь напишем сценарий для make. Для этого создаем Makefile:

**# Makefile for World project**

**binary: main.o libworld.a**

**gcc -o binary main.o -L. -lworld**

**main.o: main.c**

**gcc -c main.c**

**libworld.a: h\_world.o g\_world.o**

**ar cr libworld.a h\_world.o g\_world.o**

**h\_world.o: h\_world.c**

**gcc -c h\_world.c**

**g\_world.o: g\_world.c**

**gcc -c g\_world.c**

**clean:**

**rm -f \*.o \*.a binary**

Не забывайте ставить табуляции перед каждым правилом в целевых связках.

Собираем программу:

$ make

gcc -c main.c

gcc -c h\_world.c

gcc -c g\_world.c

ar cr libworld.a h\_world.o g\_world.o

gcc -o binary main.o -L. -lworld

$

Осталось только проверить, работает ли программа и разобраться, что же мы такое сделали:

$ ./binary

Hello World

Goodbye World

$

Итак, в приведенном примере появились три новые вещи: опции **-l** и **-L** компилятора, а также команда **ar**. Начнем с последней. Как вы уже догадались, команда ar создает статическую библиотеку (архив). В нашем случае два объектных файла объединяются в один файл libworld.a. В Linux практически все библиотеки имеют префикс lib.

Как уже говорилось, компилятор gcc сам вызывает линковщик, когда это нужно. Опция -l, переданная компилятору, обрабатывается и посылается линковщику для того, чтобы тот подключил к бинарнику библиотеку. Как вы уже заметили, у имени библиотеки "обрублены" префикс и суффикс. Это делается для того, чтобы создать "видимое безразличие" между статическими и динамическими библиотеками. Но об этом речь пойдет в других главах книги. Сейчас важно знать лишь то, что и библиотека libfoo.so и библиотека libfoo.a подключаются к проекту опцией **-lfoo**. В нашем случае libworld.a "урезалось" до -lworld.

Опция -L указывает линковщику, где ему искать библиотеку. В случае, если библиотека располагается в каталоге /lib или /usr/lib, то вопрос отпадает сам собой и опция -L не требуется. В нашем случае библиотека находится в репозитории (в текущем каталоге). По умолчанию линковщик не просматривает текущий каталог в поиске библиотеки, поэтому опция -L. (точка означает текущий каталог) необходима.

### 3.3. Пример совместно используемой библиотеки

Для того, чтобы создать и использовать динамическую (совместно используемую) библиотеку, достаточно переделать в нашем проекте Makefile.

**# Makefile for World project**

**binary: main.o libworld.so**

**gcc -o binary main.o -L. -lworld -Wl,-rpath,.**

**main.o: main.c**

**gcc -c main.c**

**libworld.so: h\_world.o g\_world.o**

**gcc -shared -o libworld.so h\_world.o g\_world.o**

**h\_world.o: h\_world.c**

**gcc -c -fPIC h\_world.c**

**g\_world.o: g\_world.c**

**gcc -c -fPIC g\_world.c**

**clean:**

**rm -f \*.o \*.so binary**

Внешне ничего не изменилось: программа компилируется, запускается и выполняет те же самые действия, что и в предыдущем случае. Изменилась **внутренняя суть**, которая играет для программиста первоочередную роль. Рассмотрим все по порядку.

Правило для сборки binary теперь содержит пугающую опцию **-Wl,-rpath,.** Ничего страшного тут нет. Как уже неоднократно говорилось, компилятор gcc сам вызывает линковщик ld, когда это надо и передает ему нужные параметры сборки, избавляя нас от ненужной платформенно-зависимой волокиты. Но иногда мы все-таки должны вмешаться в этот процесс и передать линковщику "свою" опцию. Для этого используется опция компилятора **-Wl,option,optargs,...** Расшифровываю: передать линковщику (-Wl) опцию option с аргументами optargs. В нашем случае мы передаем линковщику опцию -rpath с аргументом . (точка, текущий каталог). Возникает вопрос: что означает опция -rpath? Как уже говорилось, линковщик ищет библиотеки в определенных местах; обычно это каталоги /lib и /usr/lib, иногда /usr/local/lib. Опция -rpath просто добавляет к этому списку еще один каталог. В нашем случае это текущий каталог. Без указания опции -rpath, линковщик "молча" соберет программу, но при запуске нас будет ждать сюрприз: программа не запустится из-за отсутствия библиотеки. Попробуйте убрать опцию -Wl,-rpath,. из Makefile и пересоберите проект. При попытке запуска программа binary завершится с кодом возврата 127 (о кодах возврата будет рассказано в последующих главах). То же самое произойдет, если вызвать программу из другого каталога. Верните обратно -Wl,-rpath,., пересоберите проект, поднимитесь на уровень выше командой cd .. и попробуйте запустить бинарник командой world/binary. Ничего не получится, поскольку в новом текущем каталоге библиотеки нет.

Есть один способ не передавать линковщику дополнительных опций при помощи -Wl - это использование переменной окружения LD\_LIBRARY\_PATH. В последующих главах мы будем подробно касаться темы окружения (environment). Сейчас лишь скажу, что у каждого пользователя есть так называемое окружение (environment) представляющее собой набор пар ПЕРЕМЕННАЯ=ЗНАЧЕНИЕ, используемых программами. Чтобы посмотреть окружение, достаточно набрать команду env. Чтобы добавить в окружение переменную, достаточно набрать export ПЕРЕМЕННАЯ=ЗНАЧЕНИЕ, а чтобы удалить переменную из окружения, надо набрать export -n ПЕРЕМЕННАЯ. Будьте внимательны: export - это **внутреннаяя команда оболочки BASH**; в других оболочках (csh, ksh, ...) используются другие команды для работы с окружением. Переменная окружения LD\_LIBRARY\_PATH содержит список дополнительных "мест", разделенных двоеточиеями, где линковщих должен искать библиотеку.

Не смотря на наличие двух механизмов передачи информации о нестандартном расположении библиотек, лучше помещать библиотеки в конечных проектах в /lib и в /usr/lib. Допускается расположение библиотек в подкаталоги /usr/lib и в /usr/local/lib (с указанем -Wl,-rpath). Но заставлять конечного пользователя устанавливать LD\_LIBRARY\_PATH почти всегда является плохим стилем программирования.

Следующая немаловажная деталь - это процесс создания самой библиотеки. Статические библиотеки создаются при помощи архиватора ar, а совместно используемые - при помощи gcc с опцией -shared. В данном случае gcc опять же вызывает линковщик, но не для сборки бинарника, а для создания динамической библиотеки.

Последнее отличие - опциии -fPIC (-fpic) при компиляции h\_world.c и g\_world.c. Эта опция сообщает компилятору, что объектные файлы, полученные в результате компиляции должны содержать **позиционно-независимый код** (PIC - Position Independent Code), который используется в динамических библиотеках. В таком коде используются не фиксированные позиции (адреса), а плавающие, благодаря чему код из библиотеки имеет возможность подключаться к программе в момент запуска.

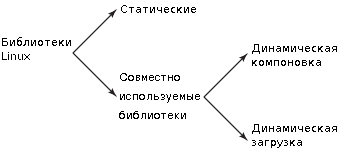
[Facebook](http://www.facebook.com/sharer.php?u=http%3A%2F%2Fwww.ibm.com%2Fdeveloperworks%2Fru%2Flibrary%2Fl-dynamic-libraries%2Findex.html&t=%D0%90%D0%BD%D0%B0%D1%82%D0%BE%D0%BC%D0%B8%D1%8F%20%D0%B4%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D1%85%20%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%20Linux)[Twitter](https://twitter.com/intent/tweet?url=https%3A%2F%2Fibm.co%2F2ztwwRa&text=%D0%90%D0%BD%D0%B0%D1%82%D0%BE%D0%BC%D0%B8%D1%8F%20%D0%B4%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D1%85%20%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%20Linux&via=developerWorks)[Linked In](http://www.linkedin.com/shareArticle?mini=true&url=http%3A%2F%2Fwww.ibm.com%2Fdeveloperworks%2Fru%2Flibrary%2Fl-dynamic-libraries%2Findex.html&title=%D0%90%D0%BD%D0%B0%D1%82%D0%BE%D0%BC%D0%B8%D1%8F%20%D0%B4%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D1%85%20%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%20Linux)[Google+](https://plus.google.com/share?url=http%3A%2F%2Fwww.ibm.com%2Fdeveloperworks%2Fru%2Flibrary%2Fl-dynamic-libraries%2Findex.html&t=%D0%90%D0%BD%D0%B0%D1%82%D0%BE%D0%BC%D0%B8%D1%8F%20%D0%B4%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D1%85%20%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%20Linux)[Отправить эту страницу по электронной почте](mailto:?subject=%D0%90%D0%BD%D0%B0%D1%82%D0%BE%D0%BC%D0%B8%D1%8F%20%D0%B4%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D1%85%20%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%20Linux&body=http%3A%2F%2Fwww.ibm.com%2Fdeveloperworks%2Fru%2Flibrary%2Fl-dynamic-libraries%2Findex.html)

[Comments](https://www.ibm.com/developerworks/ru/library/l-dynamic-libraries/#icomments)

[1](https://www.ibm.com/developerworks/ru/library/l-dynamic-libraries/" \l "icomments)

Библиотеки были придуманы для объединения схожей функциональности в отдельные модули, которые могли использоваться совместно несколькими разработчиками. Такой подход соответствует **модульному программированию**, при котором программы строятся на основе модулей. В Linux доступно два вида библиотек, и каждый из них имеет свои преимущества и недостатки. При использовании **статических библиотек** их функциональность внедряется в программный код на этапе компиляции. Напротив, **динамические библиотеки**загружаются после запуска приложения, а связывание происходит на этапе выполнения. На рисунке 1 иерархически показаны виды библиотек в Linux.

##### **Рисунок 1. Иерархия библиотек в Linux**



##### Другие статьи Тима серии на developerWorks

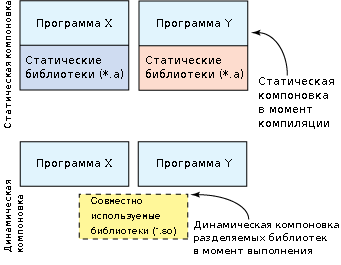
* [Анатомия загружаемых модулей ядра Linux](http://www.ibm.com/developerworks/ru/library/l-lkm/index.html)
* [Анатомия журналируемых файловых систем Linux](http://www.ibm.com/developerworks/ru/library/l-journaling-filesystems/index.html)
* [Анатомия файловых систем Linux для флэш-носителей](http://www.ibm.com/developerworks/ru/library/l-flash-filesystems/index.html)
* [Анатомия SELinux](http://www.ibm.com/developerworks/ru/library/linux/l-selinux/index.html)
* [Анатомия Linux-архитектур реального времени](http://www.ibm.com/developerworks/ru/library/l-real-time-linux/index.html)
* [Анатомия подсистемы SCSI в Linux (EN)](http://www.ibm.com/developerworks/linux/library/l-scsi-subsystem/)
* [Анатомия файловой системы Linux (EN)](http://www.ibm.com/developerworks/ru/library/l-linux-filesystem/)
* [натомия сетевого стека Linux (EN)](http://www.ibm.com/developerworks/ru/library/l-linux-kernel/)
* [Анатомия распределителя памяти slab в Linux (EN)](http://www.ibm.com/developerworks/ru/library/l-linux-slab-allocator/)
* [Анатомия методов синхронизации Linux (EN)](http://www.ibm.com/developerworks/ru/library/l-linux-synchronization/)
* [Все статьи Тима серии Анатомия ...](http://www.ibm.com/developerworks/views/linux/libraryview.jsp?topic_by=All+topics+and+related+products&sort_order=desc&lcl_sort_order=desc&search_by=anatomy&search_flag=true&type_by=Articles&show_abstract=true&start_no=1&sort_by=Date&end_no=100&show_all=false)
* [Все статьи Тима на developerWorks](http://www.ibm.com/developerworks/views/linux/libraryview.jsp?topic_by=All+topics+and+related+products&sort_order=desc&lcl_sort_order=desc&search_by=tim+jones&search_flag=true&type_by=Articles&show_abstract=false&sort_by=Date&end_no=100&show_all=false)

Существует два способа использования совместно используемых библиотек: динамическая компоновка в момент загрузки и динамическая загрузка с подключением программным путем. В статье будут описаны оба подхода.

В простых программах с минимальной функциональностью статические библиотеки могут быть предпочтительнее. В программах же, использующих несколько библиотек, применение совместно используемых библиотек позволяет снизить потребление оперативной и дисковой памяти во время работы приложения. Это достигается за счет того, что одна совместно используемая библиотека может использоваться одновременно несколькими приложениями, при этом она присутствует в памяти в единственном экземпляре. В случае со статическими библиотеками каждая программа загружает свою собственную копию библиотечных функций.

В GNU/Linux доступно два метода работы с совместно используемыми библиотеками (оба метода берут свое начало в Sun Solaris). Первый способ – это динамическая компоновка вашего приложения с совместно используемой библиотекой. При этом загрузку библиотеки при запуске программы возьмет на себя Linux (если, конечно, она не была загружена в память раньше). Второй способ подразумевает явный вызов функций библиотеки в процессе т. н. **динамической загрузки**. В этом случае программа явно загружает нужную библиотеку, а затем вызывает определенную библиотечную функцию. На этом методе обычно основан механизм загрузки подключаемых программных модулей – плагинов. Оба рассматриваемых способа показаны на рисунке 2.

##### **Рисунок 2. Сравнение статической и динамической компоновки**



## Динамическая компоновка в Linux

Рассмотрим подробнее процесс использования динамически компонуемых совместно используемых библиотек Linux. Приложение, которое запускает пользователь, представляет собой ELF-образ (Executable and Linking Format, формат исполняемых и компонуемых файлов). После запуска ядро вначале загружает образ программы в виртуальное адресное пространство создаваемого процесса; при этом анализируется ELF-секция под названием .interp, которая указывает, какой динамический загрузчик будет использоваться (как правило, это /lib/ld-linux.so). Содержимое этой секции представлено в листинге 1. Все это очень похоже на то, как в shell-скриптах первой строкой задается интерпретатор, который будет исполнять скрипт: #!/bin/sh.

##### **Листинг 1. Использование утилиты readelf для вывода заголовков исполняемого файла**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | mtj@camus:~/dl$ readelf -l dl    Elf file type is EXEC (Executable file)  Entry point 0x8048618  There are 7 program headers, starting at offset 52    Тип Elf файла – EXEC (исполняемый файл)  Точка входа 0x8048618  Имеется 7 заголовков, начиная со смещения 52    Program Headers (Заголовки программы):    Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align    (Тип)          (Смещ.)  (Вирт.адр.)(Физ.адр.) (Разм.файла)(Разм.пам.)(Флаги)(Выравн.)    PHDR           0x000034 0x08048034 0x08048034 0x000e0 0x000e0 R E 0x4    INTERP         0x000114 0x08048114 0x08048114 0x00013 0x00013 R   0x1        [Requesting program interpreter: /lib/ld-linux.so.2]        [Запрашиваемый интерпретатор программы: /lib/ld-linux.so.2]    LOAD           0x000000 0x08048000 0x08048000 0x00958 0x00958 R E 0x1000    LOAD           0x000958 0x08049958 0x08049958 0x00120 0x00128 RW  0x1000    DYNAMIC        0x00096c 0x0804996c 0x0804996c 0x000d0 0x000d0 RW  0x4    NOTE           0x000128 0x08048128 0x08048128 0x00020 0x00020 R   0x4    GNU\_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4      ...    mtj@camus:~dl$ |

Кстати, ld-linux.so тоже является совместно используемой ELF-библиотекой, хотя собрана она статически и не имеет других зависимостей. В случае использования динамической компоновки ядро передает управление на динамический компоновщик (другое название – ELF-интерпретатор), который после собственной инициализации загружает указанные совместно используемые библиотеки (если они уже не в памяти). Далее динамический компоновщик производит необходимые перемещения (relocations), включая совместно используемые объекты, на которые ссылаются требуемые совместно используемые библиотеки. Путь, по которому система будет искать совместно используемые объекты, задается переменной среды LD\_LIBRARY\_PATH. Закончив с библиотеками, компоновщик отдает управление исходной программе, которая начинает выполнение.

В основе процесса перемещения (relocation) лежит косвенная адресация, которую обеспечивают две таблицы – глобальная таблица смещений (Global Offset Table, GOT) и таблица связывания процедур (Procedure Linkage Table, PLT). В этих таблицах содержатся адреса внешних функций и данных, которые ld-linux.so должен загрузить в процессе перемещения. Получается, что код, содержащий обращение к внешним функциям и, таким образом, ссылающийся на данные этих таблиц, остается неизменным – модифицировать требуется только таблицы. Перемещение может проходить либо сразу во время загрузки программы, либо когда понадобится нужная функция. (Эта альтернатива будет рассмотрена подробно в разделе [динамическая загрузка в Linux](https://www.ibm.com/developerworks/ru/library/l-dynamic-libraries/#dynamicloading).

По завершении перемещения динамический компоновщик исполняет стартовый код каждой совместно используемой библиотеки (если этот код имеется), содержащий инициализацию и подготовку внутренних данных. Стартовый код определяется в секции .init ELF-файла. Во время выгрузки библиотеки может выполняться также и завершающий код, определяемый в секции .fini. Вызвав функции инициализации, динамический компоновщик отдает управление исходному исполняемому образу.

## Динамическая загрузка в Linux

Наряду с автоматической загрузкой и компоновкой программы с ее библиотеками есть возможность переложить эту задачу на "плечи" самой программы – это и называется динамической загрузкой. В этом случае приложение само "решает", какие библиотеки загрузить, после чего вызывает библиотечные функции, как если бы они были частью исходной программы. Однако как вы уже поняли, библиотека, отвечающая за динамическую загрузку, – это обычная совместно используемая библиотека в формате ELF. Фактически в этом процессе опять же участвует динамический компоновщик ld-linux, являющийся загрузчиком и интерпретатором ELF-файлов.

Для реализации динамической загрузки существует интерфейс динамической загрузки (Dynamic Loading API), дающий приложению пользователя возможность использовать совместно используемые библиотеки. Этот интерфейс невелик, однако он реализует все необходимое, беря всю "черную" работу на себя. Все функции интерфейса приведены в таблице 1.

##### **Таблица 1. Полный интерфейс динамической загрузки**

| **Функция** | **Описание** |
| --- | --- |
| **dlopen** | Дает программе доступ к ELF-библиотеке |
| **dlsym** | Возвращает адрес функции из библиотеки, загруженной при помощи dlopen |
| **dlerror** | Возвращает текстовое описание последней возникшей ошибки |
| **dlclose** | Закрывает доступ к библиотеке |

Вначале приложение вызывает dlopen, передавая в параметрах имя файла и режим. Функция возвращает дескриптор, который будет использоваться в дальнейшем. Режим указывает компоновщику, когда производить перемещение. Возможные варианты – RTLD\_NOW (сделать все необходимые перемещения в момент вызова dlopen) и RTLD\_LAZY (перемещения по требованию). В последнем случае работают внутренние механизмы, при которых каждое первое обращение к библиотечной функции перенаправляется динамическому компоновщику и происходит перемещение. Последующее обращение к той же функции уже не требует повторного перемещения.

Есть еще две опции режима, которые можно совместить с предыдущими путем логического ИЛИ. RTLD\_LOCAL означает, что символы данной совместно используемой библиотеки не будут доступны из других ELF-файлов, относящихся к нашему приложению. Если же такой доступ нужен (например, чтобы иметь доступ к символам главной программы из совместно используемой библиотеки), используйте флаг RTLD\_GLOBAL.

При вызове dlopen происходит автоматическое разрешение зависимостей между библиотеками. Это значит, что если некая библиотека использует другую библиотеку, функция загрузит и ее. dlopen возвращает дескриптор, используемый для дальнейшей работы с библиотекой. Прототип функции выглядит так:

|  |  |
| --- | --- |
| 1  2  3 | #include <dlfcn.h>    void \*dlopen( const char \*file, int mode ); |

По дескриптору с помощью функции dlsym находятся адреса символов библиотеки. Функция принимает в качестве параметра дескриптор и строковое имя символа и возвращает искомый адрес:

|  |  |
| --- | --- |
| 1 | void \*dlsym( void \*restrict handle, const char \*restrict name ); |

Если при работе этих функций возникла ошибка, ее текстовую формулировку можно получить при помощи dlerror. Эта функция не имеет входных аргументов и возвращает строку, если ошибка была, и NULL, если ошибки не было:

|  |  |
| --- | --- |
| 1 | char \*dlerror(); |

Если работа с библиотекой закончена и приложению больше не нужны ни дескриптор, ни ее функции, программист может вызвать dlclose. Система ведет счетчик ссылок на библиотеку, поэтому загрузка/выгрузка библиотеки разными приложениям не приводит к конфликту – библиотека будет в памяти до тех пор, пока хотя бы один пользователь работает с ней. Все адреса, полученные ранее при помощи dlsym, становятся недействительными.

|  |  |
| --- | --- |
| 1 | char \*dlclose( void \*handle ); |

## Пример, демонстрирующий динамическую загрузку

Изучив API динамической загрузки, предлагаю рассмотреть пример его использования. В примере мы реализуем оболочку, позволяющую оператору задавать исходное имя библиотеки, имя функции и аргумент. Другими словами, пользователь сможет вызывать любую функцию из произвольной библиотеки, не скомпонованной предварительно с приложением. Адрес функции находится посредством рассматриваемого API, после чего она вызывается с заданным аргументом и возвращает результат. Полный исходный текст примера представлен в листинге 2.

##### **Листинг 2. Утилита, использующая API динамической загрузки**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61 | #include <stdio.h>  #include <dlfcn.h>  #include <string.h>    #define MAX\_STRING      80      void invoke\_method( char \*lib, char \*method, float argument )  {    void \*dl\_handle;    float (\*func)(float);    char \*error;      /\* Открываем совместно используемую библиотеку \*/    dl\_handle = dlopen( lib, RTLD\_LAZY );    if (!dl\_handle) {      printf( "!!! %s\n", dlerror() );      return;    }      /\* Находим адрес функции в библиотеке \*/    func = dlsym( dl\_handle, method );    error = dlerror();    if (error != NULL) {      printf( "!!! %s\n", error );      return;    }      /\* Вызываем функцию по найденному адресу и печатаем результат \*/    printf("  %f\n", (\*func)(argument) );      /\* Закрываем объект \*/    dlclose( dl\_handle );      return;  }      int main( int argc, char \*argv[] )  {    char line[MAX\_STRING+1];    char lib[MAX\_STRING+1];    char method[MAX\_STRING+1];    float argument;      while (1) {        printf("> ");        line[0]=0;      fgets( line, MAX\_STRING, stdin);        if (!strncmp(line, "end", 3)) break;        sscanf( line, "%s %s %f", lib, method, &argument);        invoke\_method( lib, method, argument );      }    } |

Ниже приведена команда GCC, с помощью которой я рекомендую собрать наш пример. Опция -rdynamic указывает компоновщику включить в динамическую таблицу символов результирующего файла все символы, что позволит видеть стек вызовов при работе с dlopen. Опция -ldl означает компоновку с библиотекой libdl.

|  |  |
| --- | --- |
| 1 | gcc -rdynamic -o dl dl.c -ldl |

Вернемся к [листингу 2](https://www.ibm.com/developerworks/ru/library/l-dynamic-libraries/#list2) и функции main. Она фактически реализует интерпретатор, который воспринимает три аргумента в строке ввода – имя библиотеки, имя функции и параметр (число с плавающей точкой). Получив end, программа завершается, а во всех остальных случаях три аргумента передаются в функцию invoke\_method, использующую API динамической загрузки.

Вначале вызывается dlopen для получения доступа к объектному файлу библиотеки. Если функция вернула NULL, то файл не удалось найти, и программа завершается. В случае успеха мы получаем дескриптор библиотеки, который и будем дальше использовать. Затем мы пытаемся получить адрес указанной библиотечной функции с помощью dlsym, которая вернет либо пригодный для вызова адрес, либо NULL в случае ошибки.

После того как мы получили искомую функцию, определенную в ELF-объекте, следующим шагом является просто вызов этой функции. Сравните код этого примера и подход, соответствующий динамической компоновке из предыдущего раздела. Здесь мы приводим адрес из таблицы символов объектного файла к указателю на функцию, который затем и вызываем. При динамической компоновке мы получаем библиотечный символ (в частности, функцию), уже настроенный на правильный адрес. Хотя всю "грязную работу" может сделать за вас динамический компоновщик, именно представленный подход позволяет обеспечить максимальную гибкость при написании программ, расширяемых на стадии выполнения.

Наконец, вызвав требуемую функцию из ELF-объекта, мы разрываем с ним связь при помощи dlclose.

Пример работы с нашей программой приведен в листинге 3. Сначала мы компилируем и запускаем приложение. Затем мы вызываем несколько функций из математической библиотеки libm.so. Тем самым продемонстрирована возможность вызова программой произвольной функции, принадлежащей совместно используемой библиотеке, через механизм динамической загрузки – мощное средство для расширения функциональности приложений.

##### **Листинг 3. Пример простой программы, вызывающей библиотечные функции**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | mtj@camus:~/dl$ gcc -rdynamic -o dl dl.c -ldl  mtj@camus:~/dl$ ./dl  > libm.so cosf 0.0    1.000000  > libm.so sinf 0.0    0.000000  > libm.so tanf 1.0    1.557408  > конец  mtj@camus:~/dl$ |

## Утилиты

В Linux доступны разнообразные утилиты для вывода содержимого и анализа ELF-файлов (в том числе совместно используемых библиотек). Одна из самых полезных утилит – ldd, которая выдает список библиотек, от которых зависит данный ELF-объект. Например, для последнего примера вывод команды ldd будет выглядеть примерно так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | mtj@camus:~/dl$ ldd dl          linux-gate.so.1 =>  (0xffffe000)          libdl.so.2 => /lib/tls/i686/cmov/libdl.so.2 (0xb7fdb000)          libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7eac000)          /lib/ld-linux.so.2 (0xb7fe7000)  mtj@camus:~/dl$ |

ldd сообщила нам, что ELF-файл dl зависит от библиотек linux-gate.so (специальный псевдофайл, который отвечает за обработку системных вызовов Linux и не принадлежит файловой системе), libdl.so (API динамической загрузки), libc.so (библиотека GNU C) и ld-linux.so.2 (динамический загрузчик Linux – он присутствует всегда, когда есть зависимости от совместно используемых библиотек).

Понять и вывести содержимое ELF-файла поможет мощная команда readelf. Одна из интересных возможностей этой команды – вывод списка перемещаемых элементов файла. Например, покажем, какие символы в нашей тестовой программе из [листинга 2](https://www.ibm.com/developerworks/ru/library/l-dynamic-libraries/#list2) требуют перемещения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | mtj@camus:~/dl$ readelf -r dl    Relocation section '.rel.dyn' at offset 0x520 contains 2 entries:  (Раздел перемещения '.rel.dyn' со смещением 0x520 содержит 2 пункта:)   Offset     Info    Type            Sym.Value  Sym. Name   (Смещение) (Инфо)  (Тип)           (Знач.симв.)(Имя симв.)  08049a3c  00001806 R\_386\_GLOB\_DAT    00000000   \_\_gmon\_start\_\_  08049a78  00001405 R\_386\_COPY        08049a78   stdin    Relocation section '.rel.plt' at offset 0x530 contains 8 entries:   Offset     Info    Type            Sym.Value  Sym. Name  08049a4c  00000207 R\_386\_JUMP\_SLOT   00000000   dlsym  08049a50  00000607 R\_386\_JUMP\_SLOT   00000000   fgets  08049a54  00000b07 R\_386\_JUMP\_SLOT   00000000   dlerror  08049a58  00000c07 R\_386\_JUMP\_SLOT   00000000   \_\_libc\_start\_main  08049a5c  00000e07 R\_386\_JUMP\_SLOT   00000000   printf  08049a60  00001007 R\_386\_JUMP\_SLOT   00000000   dlclose  08049a64  00001107 R\_386\_JUMP\_SLOT   00000000   sscanf  08049a68  00001907 R\_386\_JUMP\_SLOT   00000000   dlopen  mtj@camus:~/dl$ |

Из этого списка видно, что перемещения требуют различные вызовы библиотек C (libc.so), в том числе вызовы к libdl.so. Обратите внимание на \_\_libc\_start\_main: это функция библиотеки C, которая вызывается перед main и производит всю необходимую инициализацию.

Среди других утилит упомянем objdump, которая печатает сведения об объектных файлах, и nm, служащую для вывода списка символов объектного файла (включая отладочную информацию). Кстати, можно запускать динамический компоновщик Linux прямо из командной строки, указав в параметре ELF-программу, которую требуется выполнить:

|  |  |
| --- | --- |
| 1  2  3  4 | mtj@camus:~/dl$ /lib/ld-linux.so.2 ./dl  > libm.so expf 0.0    1.000000  > |

В добавок, ld-linux.so может выдать список зависимостей для ELF-файла – точно так же, как делает команда ldd. Для этого существует опция --list. Не забывайте, что ld-linux.so является просто исполняемым приложением, которое в нужный момент запускается ядром.

**22. Библиотеки, коллекции функций, объектные файлы**

**Стандартной библиотекой языка Си** называется часть стандарта ANSI C, посвященная [заголовочным файлам](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B3%D0%BE%D0%BB%D0%BE%D0%B2%D0%BE%D1%87%D0%BD%D1%8B%D0%B9_%D1%84%D0%B0%D0%B9%D0%BB) и библиотечным подпрограммам. Является описанием реализации общих операций, таких как обработка ввода-вывода и [строк](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D0%BE%D0%BA%D0%BE%D0%B2%D1%8B%D0%B9_%D1%82%D0%B8%D0%BF), в [языке программирования Си](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F_%D0%A1%D0%B8). Стандартная библиотека языка Си — это описание [программного интерфейса](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D1%8B%D0%B9_%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81), а не настоящая библиотека, пригодная для использования в процессе [компиляции](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).

Стандартная библиотека Си состоит из 24 заголовочных файлов. Самыми известными примерами из них являются: locale.h - локализация (денежные параметры и т.д.), stdlib.h - функции, занимающиеся выделением памяти, контролем процесса выполнения программы, преобразованием типов и другие, math.h - для выполнения математических операций, stdio.h -

определения [макросов](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D0%BA%D1%80%D0%BE%D1%81), константы и объявления функций и типов, используемых для различных операций стандартного ввода и вывода.

Стандартная библиотека обычно поставляется вместе с [компилятором](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80). Так как компиляторы языка Си часто обеспечивают расширенную функциональность, не определенную стандартом [ANSI C](https://ru.wikipedia.org/wiki/ANSI_C), стандартная библиотека одного компилятора несовместима со стандартными библиотеками других компиляторов.

Также мы можем сами создавать **коллекции функций**, т.е. заголовочный файл. Это делается для того, чтобы уменьшить код программы и если у нас есть, например, повторяющая в большом количестве файлов .c функция, то удобно будет вынести ее в отдельный заголовочный файл и подключать его затем в каждом c-файле при помощи #include <blablabla.h>.

Работает так: компилятор берет текст (содержимое) .h файла и подставляет в данное место программы, а дальше оно исполняется как единое целое.

Работа компилятора си в том, чтобы превратить файл с кодом из понимаемого (иногда) человеком в нечто, понимаемое компьютером. На выходе компилятор даёт **объектный файл**, который на платформе UNIX имеет обычно расширение .o, а на windows .obj. Содержимое объектного файла это, по сути, два типа объектов

* Код, соответствующий определениям функций
* Данные, соответствующие глобальным переменным, определённым в файле (если они предварительно инициализированы, то там же хранится и их значение)

Экземпляры этих объектов будут иметь имена, связанные с ними – имена переменных или функций, определения которых привели к их генерации.

Объектный код – это последовательность (подходяще закодированных) машинных инструкций, соответствующих инструкциям на языке Си – всем этим if, while и даже goto. Все эти команды оперируют разного рода информацией, и эта информация должны быть где-то сохранена (для этого нужны переменные). Кроме того, они могут обращаться к другим кускам кода, который определён в файле.

Каждый раз, когда код обращается к функции или переменной, компилятор позволяет это делать только если он видел объявление этой переменной или функции. Объявление это обещание компилятору, что где-то в программе существует определение.

На платформе UNIX можно воспользоваться утилитой nm, чтобы узнать содержимое объектного файла.

Собираем объектный файл так: cc -g -O -c file.c

Используем утилиту nm так: nm file.o

23.

## Введение в bash

### Оболочка

Если вы уже использовали Linux ранее, то наверняка знаете, что после входа в систему вас приветствует приглашение, которое выглядит примерно так:

$

На практике приглашение, которое вы видите, может немного отличаться. Например, оно может содержать имя хоста, имя текущей рабочей директории, или все вместе. Не зависимо от того, как выглядит ваше приглашение, есть одна несомненная вещь: программа, которая выводит это приглашение, называется оболочка интерпретатора команд (от англ. shell — оболочка, он же командная строка или терминал — прим. пер.), и, вероятнее всего, вашей командной оболочкой будет 'bash'.

### А у вас запущен bash?

Вы можете убедиться, что используете bash, набрав:

$ **echo $SHELL**  
/bin/bash

Если строчка выше выдает ошибку, или ответ не соответствует, возможно, что вы запустили другую оболочку. В этом случае большая часть этого руководства все еще будет полезна, но было бы значительно лучше для вас переключиться на bash, ради подготовки к экзамену 101.

### О bash

Bash — это акроним от Bourne-again-shell, от англ. «ещё-одна-командная-оболочка-Борна» или «рождённая-вновь-командная оболочка» (тут игра слов Bourne/born — прим. пер.), и является оболочкой по умолчанию для большинства Linux-систем. Задача оболочки получать от вас команды, через которые вы взаимодействуете с Linux-системой. После того, как вы закончили ввод команд, вы можете выйти из оболочки (exit) или закончить сеанс (logout), в этом случае вы увидите приглашение входа в систему.

Кстати, вы также можете выйти из оболочки bash нажав control-D в приглашении.

### Использование «cd»

Вы, возможно, уже обнаружили, что пялиться на приглашение bash — не самое впечатляющее занятие в мире. Ну что ж, давайте узнаем как путешествовать по нашей файловой системе. В приглашении введите пожалуйста следующую команду (без $):

$ **cd /**

Вы только что сообщили bash, что хотите работать в директории /, также известной, как корневая; все директории в системе имеют форму дерева, и / является его вершиной, т.е. корнем (в информатике деревья растут наоборот, корень вверху, а ветки спускаются вниз — прим. пер.). cd устанавливает директорию, в которой вы в данный момент работаете, также известную как «текущая рабочая директория».

### Пути

Чтобы узнать текущую рабочую директорию в bash нужно набрать:

$ **pwd**  
/

В примере с cd, аргумент / называется путь. Он сообщает cd куда мы хотим отправиться. В частности, аргумент / это абсолютный путь, что значит, что он задает расположение относительно корня дерева файловой системы.

### Абсолютные пути

Ниже несколько из них:

/dev  
/usr  
/usr/bin  
/usr/local/bin

Как можно заметить, у всех абсолютных путей есть одна общая черта, они начинаются с /. Указывая, допустим, */usr/local/bin* в качестве аргумента для cd, мы сообщаем, что хотим попасть в / директорию, затем в *usr* директорию внутри нее, и так далее в *local* и *bin*, вниз по дереву. Абсолютные пути всегда отсчитываются начиная от / сперва.

### Относительные пути

Другой тип пути называется «относительный путь». bash, cd, и другие команды всегда интерпретируют их относительно текущей директории. Относительные пути НИКОГДА не начинаются с /. Так, если мы сначала переместимся в */usr*:

$ **cd /usr**

То, затем мы можем использовать относительный путь *local/bin*, чтобы попасть в директорию */usr/local/bin*:

$ **cd local/bin**  
$ **pwd**  
/usr/local/bin

### Использование ..

Относительные пути могут также содержать одну или более ".." директории. Директория ".." специальная; она указывает на родительскую директорию. Так, продолжая с примера выше:

$ **pwd**  
/usr/local/bin  
$ **cd ..**  
$ **pwd**  
/usr/local

Как видно, наша текущая директория теперь */usr/local*. Мы смогли переместиться «назад» на одну директорию относительно текущей, где были до того.

Кроме того, мы также можем использовать ".." в существующем относительном пути, позволяющем нам переместиться в директорию «рядом» с той, в которой находимся:

$ **pwd**  
/usr/local  
$ **cd ../share**  
$ **pwd**  
/usr/share

### Примеры относительных путей

Относительные пути могут быть чуточку более сложными. Ниже несколько примеров, попробуйте самостоятельно догадаться, где вы окажитесь после набора каждой из этих команд.

$ **cd /bin**  
$ **cd ../usr/share/zoneinfo**  
  
$ **cd /usr/X11R6/bin**  
$ **cd ../lib/X11**  
  
$ **cd /usr/bin**  
$ **cd ../bin/../bin**

А теперь наберите их и проверьте свои предположения. ;)

### Понимание .

Перед тем как мы закончим изучение cd, есть несколько моментов, которые необходимо прояснить. Во-первых, есть еще одна специальная директория ".", которая означает «текущая директория». Хотя она и не используется с командой cd, но часто используется для выполнения программы из текущей директории, как в следующем примере:

$ **./myprog**

В данном случае будет запущена исполняемая программа *myprog*, находящаяся в текущей рабочей директории.

### cd и домашняя директория

Если бы мы хотели переместиться в нашу домашнюю директорию, то могли бы набрать:

$ **cd**

Без каких либо аргументов cd переместит в вашу домашнюю директорию, которая будет */root* для суперпользователя, или обычно */home/username* (где *username* — имя пользователя в системе — прим.пер.) для любого другого пользователя. Но, что если мы хотим указать файл в нашей домашней директории? Может быть мы хотим передать путь к файлу в качестве аргумента нашей программе *myprog*. Если файл расположен в нашей домашней директории, мы можем набрать:

$ **./myprog /home/drobbins/myfile.txt**

Однако, использования абсолютного пути вроде этого, не всегда удобно. К счастью, мы можем использовать символ ~ (тильда), чтобы проделать то же самое:

$ **./myprog ~/myfile.txt**

### Другие домашние директории пользователей

Bash воспримет одиночную ~ как указатель на вашу домашнюю директорию, но вы также можете использовать её для указания на домашние директории других пользователей. Например, если мы хотели сослаться на файл под названием *fredsfile.txt* в домашней директории пользователя *fred*, то могли бы набрать:

$ **./myprog ~fred/fredsfile.txt**

В данном отрывке рассмотрены команды работы с файловой системой и файлами, такие как ls, mkdir, touch, mv, cp, затронуто использование echo и cat. Рассказано про иноды файловой системы.

## Использование команд Linux

### Знакомство с ls

А сейчас, мы быстренько пройдемся по команде ls. Скорее всего вы уже хорошо знакомы с этой командой, и знаете, что набрав ls получите список содержимого текущей рабочей директории:

$ **cd /usr**

$ **ls**

X11R6 doc i686-pc-linux-gnu lib man sbin ssl

bin gentoo-x86 include libexec portage share tmp

distfiles i686-linux info local portage.old src

Указав опцию -a, вы можете увидеть полный список, включая скрытые файлы и директории, начинающиеся с ".". Как видно в следующем примере, ls -a выводит также особые связывающие директории "." и "..":

$ **ls -a**

. bin gentoo-x86 include libexec portage share tmp

.. distfiles i686-linux info local portage.old src

X11R6 doc i686-pc-linux-gnu lib man sbin ssl

### Развернутые списки директорий

Вы также можете задать одну и более директорий или файлов в командной строке с ls. Если вы укажите файл, то ls покажет вам только этот файл. А если зададите директорию, то ls выдаст ее содержимое. Опция -l очень удобна, когда необходимо посмотреть права доступа, владельца, время последнего изменения и размер в списке содержимого директории.

В следующем примере мы использовали опцию -l чтобы отобразить содержимое моей директории */usr*:

$ **ls -l /usr**

drwxr-xr-x 7 root root 168 Nov 24 14:02 X11R6

drwxr-xr-x 2 root root 14576 Dec 27 08:56 bin

drwxr-xr-x 2 root root 8856 Dec 26 12:47 distfiles

lrwxrwxrwx 1 root root 9 Dec 22 20:57 doc -> share/doc

drwxr-xr-x 62 root root 1856 Dec 27 15:54 gentoo-x86

drwxr-xr-x 4 root root 152 Dec 12 23:10 i686-linux

drwxr-xr-x 4 root root 96 Nov 24 13:17 i686-pc-linux-gnu

drwxr-xr-x 54 root root 5992 Dec 24 22:30 include

lrwxrwxrwx 1 root root 10 Dec 22 20:57 info -> share/info

drwxr-xr-x 28 root root 13552 Dec 26 00:31 lib

drwxr-xr-x 3 root root 72 Nov 25 00:34 libexec

drwxr-xr-x 8 root root 240 Dec 22 20:57 local

lrwxrwxrwx 1 root root 9 Dec 22 20:57 man -> share/man

lrwxrwxrwx 1 root root 11 Dec 8 07:59 portage -> gentoo-x86/

drwxr-xr-x 60 root root 1864 Dec 8 07:55 portage.old

drwxr-xr-x 3 root root 3096 Dec 22 20:57 sbin

drwxr-xr-x 46 root root 1144 Dec 24 15:32 share

drwxr-xr-x 8 root root 328 Dec 26 00:07 src

drwxr-xr-x 6 root root 176 Nov 24 14:25 ssl

lrwxrwxrwx 1 root root 10 Dec 22 20:57 tmp -> ../var/tmp

Первая колонка показывает информацию о правах доступа для каждого элемента. Чуть позже я объясню, как её интерпретировать. Следующая колонка содержит числа ссылок на каждый элемент файловой системы, позже мы вернемся к этому. Третья и четвертая колонки — это список владельцев и групп, соответственно. Пятая колонка — размер объекта. Шестая — время последнего изменения (mtime) объекта. И наконец, последняя колонка с именами объектов. Если файлы являются символическими ссылками, то вы увидите стрелку -> и путь, куда указывает эта символическая ссылка.

### Смотрим на директории

Иногда вы захотите взглянуть на директорию, а не внутрь нее. В этом случае вы можете указать опцию -d, которая скажет ls рассматривать любую директорию, как внутреннюю:

$ **ls -dl /usr /usr/bin /usr/X11R6/bin ../share**

drwxr-xr-x 4 root root 96 Dec 18 18:17 ../share

drwxr-xr-x 17 root root 576 Dec 24 09:03 /usr

drwxr-xr-x 2 root root 3192 Dec 26 12:52 /usr/X11R6/bin

drwxr-xr-x 2 root root 14576 Dec 27 08:56 /usr/bin

### Рекурсивный и инодный списки

Так вы можете использовать -d чтобы смотреть на директорию, но также можно использовать -R для противоположного: не только лишь глянуть внутрь директории, но и рекурсивно посмотреть все директории с файлами внутри нее! Мы не включим в руководство никакого примера вывода для этой опции (поскольку обычно он очень объемный), но возможно вы захотите попробовать несколько команд ls -R и ls -Rl, чтобы почувствовать как это работает.

Наконец, опция -i может использоваться для отображения числа инодов для объектов в списке файловой системы:

$ **ls -i /usr**

1409 X11R6 314258 i686-linux 43090 libexec 13394 sbin

1417 bin 1513 i686-pc-linux-gnu 5120 local 13408 share

8316 distfiles 1517 include 776 man 23779 src

43 doc 1386 info 93892 portage 36737 ssl

70744 gentoo-x86 1585 lib 5132 portage.old 784 tmp

### Понятие инода

Каждому объекту файловой системы назначен уникальный индекс, называемый номером инода. Это может показаться банальным, но понятие инодов очень важно для понимания большинства операций в файловой системе. Рассмотрим например ссылки "." и "..", которые появляются в каждой директории. Чтобы полностью понять, чем на самом деле является директория "..", мы сперва взглянем на номер инода у */usr/local*:

$ **ls -id /usr/local**  
5120 /usr/local

У директории */usr/local* номер инода равен 5120. А теперь посмотрим номер инода у */usr/local/bin/..*:

$ **ls -id /usr/local/bin/..**  
5120 /usr/local/bin/..

Как видно, директория */usr/local/bin/..* имеет такой же номер, как у */usr/local*! Посмотрим, как можно справиться с этим шокирующим откровением. В прошлом мы полагали, что */usr/local* сама является директорией. Теперь же, мы обнаружили, что фактически директория — это инод с номером 5120, и нашли, по меньшей мере, два элемента (называемых «ссылками»), которые указывают на данный инод. И */usr/local*, и */usr/local/bin/..* — ссылки на 5120-ый инод. Хотя этот инод и существует только в одном месте на диске, тем не менее на него может быть множество ссылок.

На самом деле, мы даже можем увидеть общее количество ссылок ведущих на этот, 5120 инод, используя команду ls -dl:

$ **ls -dl /usr/local**

drwxr-xr-x 8 root root 240 Dec 22 20:57 /usr/local

Если взглянуть на вторую колонку слева, то видно, что на директорию */usr/local* (инод 5120) ссылаются восемь раз. На моей системе на этот инод ведут следующие пути:

/usr/local  
/usr/local/.  
/usr/local/bin/..  
/usr/local/games/..  
/usr/local/lib/..  
/usr/local/sbin/..  
/usr/local/share/..  
/usr/local/src/..

### mkdir

Давайте быстренько пройдемся по команде mkdir, которая используется для создания новых директорий. Следующий пример создает три новых директории, *tic*, *tac*, и *toe*, все внутри */tmp*:

$ **cd /tmp**  
$ **mkdir tic tac toe**

По умолчанию, команда mkdir не создает для вас родительские директории; весь путь вплоть до последнего (создаваемого) элемента должен существовать. Так, если вы захотите создать вложенные директории *won/der/ful*, вам придется выполнить три отдельные команды mkdir:

$ **mkdir won/der/ful**  
mkdir: cannot create directory `won/der/ful': No such file or directory  
$ **mkdir won**  
$ **mkdir won/der**  
$ **mkdir won/der/ful**

Однако, у mkdir есть очень удобная опция -p, которая говорит mkdir создавать любые отсутствующие родительские директории, как можете увидеть тут:

$ **mkdir -p easy/as/pie**

В целом очень просто. Чтобы узнать больше о команде mkdir наберите man mkdir и прочитайте инструкцию. Это же касается почти всех команд, рассмотренных здесь (например man ls), исключая cd, которая встроена в bash.

### touch

Сейчас мы собираемся окинуть взглядом команды cp и mv, используемые для копирования, переименования и перемещения файлов и директорий. Но начнем обзор воспользовавшись командой touch, чтобы создать файл в */tmp*:

$ **cd /tmp**  
$ **touch copyme**

Команда touch обновляет «mtime» (время последней модификации — прим. пер.) файла, если тот существует (вспомните шестую колонку в выводе ls -l). Если файл не существует, то новый, пустой файл будет создан. Сейчас у вас должен быть файл */tmp/copyme* с нулевым размером.

### echo

Теперь, когда файл существует, давайте добавим немного данных в него. Можно сделать это с помощью команды echo, которая принимает аргументы и печатает их на стандартный вывод. Сперва, команда echo сама по себе:

$ **echo "firstfile"**  
firstfile

А сейчас, та же команда echo, но с перенаправлением вывода:

$ **echo "firstfile" > copyme**

Знак «больше» сообщает оболочке записывать вывод echo в файл по имени *copyme*. Этот файл будет создан, если не существовал, или перезаписан, если существует. Набрав ls -l, увидим, что файл copyme имеет размер в 10 байт, так как содержит слово firstfile и символ новой строки:

$ **ls -l copyme**

-rw-r--r-- 1 root root 10 Dec 28 14:13 copyme

### cat и cp

Чтобы вывести содержимое файла на терминал, используйте команду cat:

$ **cat copyme**  
firstfile

Сейчас, мы можем воспользоваться основным вызовом команды cp для создания файла *copiedme* из оригинального *copyme*:

$ **cp copyme copiedme**

Ниже проверим, что это действительно разные файлы; у них отличаются номера инодов:

$ **ls -i copyme copiedme**

648284 copiedme 650704 copyme

### mv

А сейчас давайте воспользуемся командой mv для переименования *copiedme* в *movedme*. Номер иноды останется прежний; однако, имя файла, указывающее на инод, изменится.

$ **mv copiedme movedme**

$ **ls -i movedme**

648284 movedme

Номер инода у перемещаемого файла остается прежним до тех пор, пока файл назначения находится в той же файловой системе, что и исходный файл. Мы подробнее рассмотрим файловую систему в третьей части нашего руководства.

Пока мы рассказываем об mv, давайте посмотрим, как еще можно использовать эту команду. mv, помимо возможности переименовать файлы, позволяет перемещать один или более файлов в другое место в иерархии директорий. Например, чтобы переместить */var/tmp/myfile.txt* в директорию */home/drobbins* (которая является моей домашней), я наберу (а мог бы воспользоваться ~ — прим. пер.):

$ **mv /var/tmp/myfile.txt /home/drobbins**

После этого *myfile.txt* будет перемещен в */home/drobbins/myfile.txt*. И если */home/drobbins* располагается в другой файловой системе, нежели */var/tmp*, команда mv скопирует *myfile.txt* в новую файловую систему и удалит его из старой. Как вы уже могли догадаться, когда *myfile.txt* перемещается между файловыми системами, то *myfile.txt* на новом месте получает новый номер инода. Это все потому, что у каждой файловой системы свой независимый набор номеров инодов.

Мы также можем воспользоваться mv для перемещения нескольких файлов в одну директорию. К примеру, чтобы переместить *myfile1.txt* и *myarticle3.txt* в */home/drobbins*, потребуется набрать:

$ **mv /var/tmp/myfile1.txt /var/tmp/myarticle3.txt /home/drobbins**

## Создание ссылок и удаление файлов

### Жесткие ссылки

Мы уже упоминали термин «ссылка», когда рассказывали о взаимоотношениях между директориями (их именами) и инодами (индексным номерами, лежащими в основе файловой системы, которых мы не замечаем). Вообще в Linux существует два типа ссылок. Тип, о котором мы уже говорили ранее, называется «жесткие ссылки». Каждый инод может иметь произвольное число жестких ссылок. Когда уничтожается последняя жесткая ссылка, и не одна программа не держит файл открытым, то Linux автоматически удаляет его. Новые жесткие ссылки можно создать воспользовавшись командой ln:

$ **cd /tmp**

$ **touch firstlink**

$ **ln firstlink secondlink**

$ **ls -i firstlink secondlink**

15782 firstlink 15782 secondlink

Как видите, жесткие ссылки работают на уровне инодов, для указания конкретного файла. В Linux системах, для жестких ссылок есть несколько ограничений. В частности, можно создавать жесткие ссылки только на файлы, не на директории. Да-да, именно так; хотя "." и ".." являются созданными системой жесткими ссылками на директории, вам (даже от имени пользователя «root») не разрешается создавать любые свои собственные. Второе ограничение жестких ссылок состоит в том, что нельзя связать ими несколько файловых систем. Это значит, что у вас не получится создать жесткую ссылку с */usr/bin/bash* на */bin/bash* и если ваши директории */* и */usr* находятся в разных файловых системах (разделах — прим. пер.).

### Символьные ссылки

В практике, символьные ссылки (или символические, иногда «симлинки» — от англ.) используются гораздо чаще, чем жесткие. Симлинки — это файлы особого типа, которые ссылаются на другие файлы по имени, а не прямо по номеру инода. Они не спасают файлы от удаления; если файл, на который указывает ссылка, исчезает, то симлинк перестает работать, ломается.

Символические ссылки можно создать передав для ln опцию -s.

$ **ln -s secondlink thirdlink**

$ **ls -l firstlink secondlink thirdlink**

-rw-rw-r-- 2 agriffis agriffis 0 Dec 31 19:08 firstlink

-rw-rw-r-- 2 agriffis agriffis 0 Dec 31 19:08 secondlink

lrwxrwxrwx 1 agriffis agriffis 10 Dec 31 19:39 thirdlink -> secondlink

В выводе ls -l символьные ссылки можно отличить тремя способами. Во-первых, обратите внимание на символ l в первой колонке. Во-вторых, размер символической ссылки равен количеству символов в ней (*secondlink* в нашем случае). В-третьих, последняя колонка в выводе показывает куда ведет ссылка с помощью интуитивного обозначения "->".

### Симлинки детально

Символические ссылки в целом более гибкие, чем жесткие. Вы можете создавать символьные ссылки на любой объект файловой системы, включая директории. И благодаря тому, что их реализация основана на путях (не инодах), можно совершенно свободно создать символьную ссылку указывающую на объект другой файловой системы. Однако, сей факт также делает их сложными в понимании.

Предположим, что мы хотим создать ссылку в */tmp*, которая указывает на */usr/local/bin*. Нам следует набрать:

$ **ln -s /usr/local/bin bin1**

$ **ls -l bin1**

lrwxrwxrwx 1 root root 14 Jan 1 15:42 bin1 -> /usr/local/bin

Либо, альтернативный вариант:

$ **ln -s ../usr/local/bin bin2**

$ **ls -l bin2**

lrwxrwxrwx 1 root root 16 Jan 1 15:43 bin2 -> ../usr/local/bin

Как вы видите, обе символические ссылки указывают на одну директорию. Однако, если наша вторая символьная ссылка когда-нибудь будет перемещена в другую директорию, то она может «поломаться» из-за относительности пути:

$ **ls -l bin2**

lrwxrwxrwx 1 root root 16 Jan 1 15:43 bin2 -> ../usr/local/bin

$ **mkdir mynewdir**  
$ **mv bin2 mynewdir**  
$ **cd mynewdir**  
$ **cd bin2**  
bash: cd: bin2: No such file or directory

Потому, что директории */tmp/usr/local/bin* не существует, мы больше не можем переместиться в *bin2*; другими словами, *bin2* сейчас сломана.

По этой причине, избегать создания ссылок с относительной информацией о пути, иногда будет хорошей идеей. Тем не менее, существует множество случаев, где относительные символические ссылки крайне удобны. Рассмотрим пример в котором мы хотим создать альтернативное имя для программы в */usr/bin*:

# **ls -l /usr/bin/keychain**

-rwxr-xr-x 1 root root 10150 Dec 12 20:09 /usr/bin/keychain

От имени суперпользователя мы хотим короткий синоним для *keychain*, такой, как *kc*. В этом примере у нас есть root-доступ, о чем свидетельствует измененное на "#" приветствие bash. Нам нужен root-доступ потому, что обычные пользователи не имеют прав создавать файлы в */usr/bin*. От имени суперпользователя мы можем создать альтернативное имя для *keychain* следующим образом:

# **cd /usr/bin**

# **ln -s /usr/bin/keychain kc**

# **ls -l keychain**

-rwxr-xr-x 1 root root 10150 Dec 12 20:09 /usr/bin/keychain

# **ls -l kc** 

lrwxrwxrwx 1 root root 17 Mar 27 17:44 kc -> /usr/bin/keychain

В этом примере мы создали символьную ссылку под названием kc, которая указывает на файл */usr/bin/keychain*.

Пока это решение будет работать, но создаст проблему, если мы решим переместить оба файла, */usr/bin/keychain* и */usr/bin/kc* в */usr/local/bin*:

# **mv /usr/bin/keychain /usr/bin/kc /usr/local/bin**

# **ls -l /usr/local/bin/keychain**

-rwxr-xr-x 1 root root 10150 Dec 12 20:09 /usr/local/bin/keychain

# **ls -l /usr/local/bin/kc**

lrwxrwxrwx 1 root root 17 Mar 27 17:44 kc -> /usr/bin/keychain

Поскольку мы использовали абсолютный путь для символической ссылки *kc*, то она все еще ссылается на */usr/bin/keychain*, которого не существует с тех пор как мы переместили */usr/bin/keychain* в */usr/local/bin*.

Это привело к тому, что симлинк kc сейчас не работает. Как относительные, так и абсолютные пути в символьных ссылках имеют свои достоинства, и, в зависимости от вашей задачи, нужно использовать соответствующий тип пути. Часто, и относительный, и абсолютный путь, будут работать одинаково хорошо. Пример ниже будет работать, даже после перемещения обоих файлов:

# **cd /usr/bin**

# **ln -s keychain kc**

# **ls -l kc**

lrwxrwxrwx 1 root root 8 Jan 5 12:40 kc -> keychain

# **mv keychain kc /usr/local/bin**  
# **ls -l /usr/local/bin/keychain**

-rwxr-xr-x 1 root root 10150 Dec 12 20:09 /usr/local/bin/keychain

# **ls -l /usr/local/bin/kc**

lrwxrwxrwx 1 root root 17 Mar 27 17:44 kc -> keychain

Теперь, мы можем запустить программу *keychain* набрав */usr/local/bin/kc*. */usr/local/bin/kc* указывает на программу *keychain* в той же директории, где находится *kc*.

### rm

Итак, мы знаем как использовать cp, mv и ln, настало время узнать о том, как можно удалять объекты из файловой системы. Обычно это делается с помощью команды rm. Чтобы удалить файлы, просто укажите их в командной строке:

$ **cd /tmp**

$ **touch file1 file2**

$ **ls -l file1 file2**

-rw-r--r-- 1 root root 0 Jan 1 16:41 file1

-rw-r--r-- 1 root root 0 Jan 1 16:41 file2

$ **rm file1 file2**  
$ **ls -l file1 file2**  
ls: file1: No such file or directory  
ls: file2: No such file or directory

Имейте ввиду, что под Linux, однажды удаленный файл, обычно исчезает на века. Поэтому многие начинающие системные администраторы используют опцию -i, когда удаляют файлы. Опция -i сообщает rm удалять файлы в интерактивном режиме — это значит спрашивать перед удалением любого файла. Например:

$ **rm -i file1 file2**  
rm: remove regular empty file `file1'? y  
rm: remove regular empty file `file2'? y

В примере выше команда rm запрашивает подтверждение на удаление каждого из указанных файлов. В случае согласия, я должен был вводить «y» и нажать enter, дважды. Если бы я ввел «n», то файл бы остался цел. Или, если я сделал что-нибудь не так, я мог бы нажать Control-C и сбросить выполнение команды rm -i целиком — всяко до того, как это могло нанести какой-нибудь ущерб моей системе.

Если вы все еще учитесь пользоваться командой rm, то может быть полезным добавить при помощи вашего любимого текстового редактора следующую строку в ваш файл *~/.bashrc*, и затем выйти (logout) и войти (login) в систему вновь. После этого, всякий раз, когда вы наберете rm, оболочка bash преобразует ее автоматически в команду rm -i. Таким образом, rm будет всегда работать в интерактивном режиме:

**alias rm="rm -i"**

### rmdir

Для удаления директорий у вас имеется два варианта. Вы можете удалить все объекты внутри директории и затем воспользоваться rmdir для удаления самой директории:

$ **mkdir mydir**  
$ **touch mydir/file1**  
$ **rm mydir/file1**  
$ **rmdir mydir**

Этот метод широко известен под названием «способ удаления директорий для лохов». Все реальные пацаны и админы-гуру съевшие ~~пользователя~~ собаку на этом деле, используют гораздо более удобную команду rm -rf, описанную далее.

Самый лучший способ удалить директорию состоит в использовании опций «рекурсивного принуждения» (recursive force) команды rm, чтобы приказать ей удалять указанную директорию, также как и объекты содержащиеся внутри:

$ **rm -rf mydir**

Обычно, rm -rf является наиболее предпочтительным методом для удаления древа директорий. Будьте очень осторожны, когда пользуетесь rm -rf, так как ее мощь может быть использована по обе стороны: добра и зла. =)

## Использование джокеров

### Знакомство с джокерами

В повседневном использовании Linux часто случается, когда вам нужно выполнить единичную операцию (например rm) на множестве объектов файловой системы за раз. В подобных ситуациях вписывать множество файлов в командную строку, зачастую, довольно обременительно:

$ rm file1 file2 file3 file4 file5 file6 file7 file8

Для решения этой проблемы, вы можете извлечь пользу из встроенной в Linux поддержки джокеров. Эта поддержка также называется «globbing» (по историческим причинам; в русском также известно как «универсализация файловых имен» — прим. пер.), позволяет указать множество файлов за раз, используя шаблон «дикой конкатенации». Bash и другие команды Linux интерпретируют этот шаблон просматривая диск в поисках файлов, которые ему удовлетворяют. Так, если у вас есть файлы *file1* до *file8* в текущей рабочей директории, то можете их удалить набрав:

$ **rm file[1-8]**

Или если просто хотите удалить все файлы с именами начинающимися с *file*, включая сам файл по имени *file*, вы можете набрать:

$ **rm file\***

Дикая конкатенация "\*" совпадает с любым символом или набором символов, или даже с их отсутствием. Разумеется, glob-джокеры возможно использовать для гораздо большего, чем простое удаление файлов, как мы увидим в дальнейшем.

### Неподходящие шаблоны

Если вам необходимо перечислить все объекты файловой системы в */etc*, начинающиеся с «g», а также сам *g*, то можно ввести:

$ **ls -d /etc/g\***

/etc/gconf /etc/ggi /etc/gimp /etc/gnome /etc/gnome-vfs-mime-magic /etc/gpm /etc/group /etc/group-

А сейчас о том, что случится если вы укажите шаблон, который не подходит ни под один объект файловой системы. В следующем примере мы попытались перебрать все файлы в */usr/bin*, которые начинаются с «asdf» и оканчиваются на «jkl», потенциально включая и *asdfjkl*:

$ **ls -d /usr/bin/asdf\*jkl**  
ls: /usr/bin/asdf\*jkl: No such file or directory

Вот что случится. Обычно, когда мы задаем шаблон, и если в него укладываются один или несколько файлов в подразумеваемой файловой системе, то bash заменяет наш шаблон на разделенный пробелами список всех подходящих объектов. Однако, когда с шаблоном нет совпадений, bash оставляет переданный аргумент с джокерами как есть. Так вот, затем ls не может найти файл */usr/bin/asdf\*jkl* и выдает нам ошибку. Основное правило тут такое: glob-шаблоны разворачиваются только если совпадают с объектами файловой системы. В противном случае, остаются не тронутыми и буквально передаются в вызов программы.

### Синтаксис джокеров: \* и ?

Так, мы уже посмотрели как работает globbing, теперь же стоит рассмотреть синтаксис джокеров. В качестве джокеров используются специальные символы:

**\*** — совпадает с нулевым или большим количеством символов. Это значит: «тут может быть все что угодно, включая и ничего». Примеры:

* **/etc/g\*** совпадает со всеми файлами в */etc*, начинающимися с *g* и самим файлом *g*;
* **/tmp/my\*1** совпадает со всеми файлами в */tmp*, которые начинаются с my и заканчиваются 1, включая файл my1.

**?** — равен любому одному символу. Примеры:

* **myfile?** совпадает с любым файлом, чье имя составляет *myfile* и следующим за этим какой-либо один символ;
* **/tmp/notes?txt** совпадет, например, с */tmp/notes.txt* и */tmp/notes\_txt*, если они существуют.

### Синтаксис джокера: []

Этот джокер похож на **?**, но более точен. Чтобы его использовать, поместите любые символы, какие вам нужны, внутрь **[]**. Полученное выражение будет удовлетворять любому одному из этих символов. Вы также можете воспользоваться "-", для указания диапазона, и даже комбинации диапазонов. Примеры:

* **myfile[12]** совпадет с *myfile1* и *myfile2*. Джокер сработает если хотя бы один из этих файлов существует в текущей директории;
* **[Cc]hange[Ll]og** совпадет с *Changelog*, *ChangeLog*, *changeLog* и *changelog*. Как можете заметить, использование скобочных джокеров очень удобно для указания вариантов с заглавными буквами;
* **ls /etc/[0-9]\*** покажет все файлы в */etc*, начинающиеся с десятичной цифры;
* **ls /tmp/[A-Za-z]\*** отобразит все файлы в */tmp*, которые начинаются с большой или маленькой латинской буквы.

Конструкция **[!]** эквивалентна конструкции [], за исключением того, что вместо совпадения с символами внутри скобок, она удовлетворяет любому символу, который НЕ перечислен между [! и ]. Пример:

* **rm myfile[!9]** удалит все файлы с названием *myfile* плюс один символ, кроме *myfile9*.

### Предостережения о джокерах

Сейчас несколько предостережений, чтобы быть осторожными во время использования джокеров. Поскольку bash обрабатывает относящиеся к джокерам символы (?, [, ] и \*) особым образом, вам надо особенно позаботиться, когда вы пишите аргумент для команды, содержащий эти символы. Например, если вы хотите создать файл, содержащий строку "[fo]\*", то следующая команда может не дать желаемого результата:

$ **echo [fo]\* > /tmp/mynewfile.txt**

Если шаблон [fo]\* совпадет с какими-либо файлами в текущей рабочей директории, то вы обнаружите их имена внутри */tmp/mynewfile.txt*, вместо ожидаемого [fo]\*. Решение? Ну, одним из них будет огородить ваши символы в одиночные кавычки, которые сообщат bash не делать никаких раскрытий джокеров в них:

$ **echo '[fo]\*' > /tmp/mynewfile.txt**

Используя этот подход, ваш новый файл будет содержать [fo]\* буквально, как и предполагалось. Также, вы можете использовать экранирующую обратную косую черту, чтобы сообщить bash, что [, ] и \* должны интерпретироваться буквально, а не как джокеры:

$ **echo \[fo\]\\* > /tmp/mynewfile.txt**

Оба подхода (одиночные кавычки и экранирующая обратная косая черта) работают с одинаковым эффектом. Поскольку мы заговорили про обработку обратной косой черты, то неплохо бы было сейчас прояснить, что на случай, если вы хотите указать \ буквально, можно либо также взять ее в одиночные кавычки, либо набрать \\ (это будет обращено в \).

#### Примечание:

Двойные кавычки работают также как и одиночные, но всё еще разрешают bash делать некоторую ограниченную обработку. Следовательно, одиночные кавычки — ваш лучший выбор, если вы действительно заинтересованны в буквальной передаче текста в команду. Получить больше информации о раскрытии джокеров можно набрав man 7 glob. Чтобы получить больше информации о кавычках в bash, наберите man 1 bash и прочитайте раздел под названием QUOTING. Если в ваши планы входит сдача экзаменов LPI, то рассматривайте это как свое домашнее задание. =)

## Подведение итогов и ссылки на другие ресурсы

### Итог

Поздравляем; Вы добрались до конца нашего обзора об основах Linux! Я надеюсь, что это помогло вам укрепить свои фундаментальные познания в Linux. Темы, которые вы здесь изучили, включая основы работы в bash, основные команды Linux, ссылки и джокеры — заложили фундамент для нашего следующего пособия об основах администрирования, в котором мы рассмотрим такие темы, как регулярные выражения, принадлежность и права доступа, управления аккаунтами пользователей, и многое другое.

Продолжая погружаться в эти учебные пособия, вы скоро достигните LPIC Level 1 сертификации от Linux Professional Institute. Говоря об этой сертификации, если это действительно то, в чем вы заинтересованы, мы рекомендуем вам изучить ресурсы из параграфа ниже, которые были внимательно подобраны к материалу освещенному в этом руководстве.

## Регулярные выражения

### Что такое «регулярное выражение»?

Регулярное выражение (по англ. regular expression, сокр. «regexp» или «regex», в отечестве иногда зовется «регулярка» — прим. пер.) — это особый синтаксис используемый для описания текстовых шаблонов. В Linux-системах регулярные выражения широко используются для поиска в тексте по шаблону, а также для операций поиска и замены на текстовых потоках.

### В сравнении с глоббингом

Как только мы начнем рассматривать регулярные выражения, возможно вы обратите внимание, что их синтаксис очень похож на синтаксис подстановки имен файлов (globbing), который мы рассматривали в первой части. Однако, не стоит заблуждаться, эта схожесть очень поверхностна. Регулярные выражения и глоббинг-шаблоны, даже когда они выглядят похоже, принципиально разные вещи.

### Простая подстрока

После этого предостережения, давайте рассмотрим самое основное в регулярных выражениях, простейшую подстроку. Для этого мы воспользуемся «grep», командой, которая сканирует содержимое файла согласно заданному регулярному выражению. grep выводит каждую строчку, которая совпадает с регулярным выражением, игнорируя остальные:

$ **grep bash /etc/passwd**

operator:x:11:0:operator:/root:/bin/bash

root:x:0:0::/root:/bin/bash

ftp:x:40:1::/home/ftp:/bin/bash

Выше, первый параметр для grep, это regex; второй — имя файла. grep считывал каждую строчку из */etc/passwd* и прикладывал на нее простую regex-подстроку «bash» в поисках совпадения. Если совпадение обнаруживалось, то grep выводил всю строку целиком; в противном случае, строка игнорировалась.

### Понимание простой подстроки

В общем случае, если вы ищите подстроку, вы просто можете указать её буквально, не используя каких-либо «специальных» символов. Вам понадобиться особо позаботиться, только если ваша подстрока содержит +, ., \*, [, ] или \, в этом случае эти символы должны быть экранированы обратным слешем, а подстрока заключаться в кавычки. Вот несколько примеров регулярных выражений в виде простой подстроки:

* /tmp (поиск строки /tmp)
* "\[box\]" (поиск строки [box])
* "\\*funny\\*" (поиск строки \*funny\*)
* «ld\.so» (поиск строки ld.so)

### Метасимволы

С помощью регулярных выражений используя метасимволы возможно осуществлять гораздо более сложный поиск, чем в примерах, которые недавно рассматривали. Один из таких метасимволов "." (точка), который совпадает с любым единичным символом:

$ **grep dev.sda /etc/fstab**

/dev/sda3 / reiserfs noatime,ro 1 1

/dev/sda1 /boot reiserfs noauto,noatime,notail 1 2

/dev/sda2 swap swap sw 0 0

#/dev/sda4 /mnt/extra reiserfs noatime,rw 1 1

В этом примере текст dev.sda не появляется буквально ни в одной из строчек из */etc/fstab*. Однако, grep сканирует его не буквально по строке dev.sda, а по dev.sda шаблону. Запомните, что "." будет соответствовать любому единичному символу. Как вы видите, метасимвол "." функционально эквивалентен тому, как работает метасимвол "?" в glob-подстановках.

### Использование []

Если мы хотим задать символ конкретнее, чем это делает ".", то можем использовать [ и ] (квадратные скобки), чтобы указать подмножество символов для сопоставления:

$ **grep dev.sda[12] /etc/fstab**

/dev/sda1 /boot reiserfs noauto,noatime,notail 1 2

/dev/sda2 swap swap sw 0 0

Как вы заметили, в частности, данная синтаксическая конструкция работает идентично конструкции "[]" при glob-подстановке имен файлов. Опять же, в этом заключается одна из неоднозначностей в изучении регулярных выражений: синтаксис похожий, но не идентичный синтаксису glob-подстановок, что сбивает с толку.

### Использование [^]

Вы можете обратить значение квадратных скобок поместив ^ сразу после [. В этому случае скобки будут соответствовать любому символу который НЕ перечислен внутри них. И опять, заметьте что [^] мы используем с регулярными выражением, а [!] с glob:

$ **grep dev.hda[^12] /etc/fstab**

/dev/hda3 / reiserfs noatime,ro 1 1

#/dev/hda4 /mnt/extra reiserfs noatime,rw 1 1

### Отличающийся синтаксис

Очень важно отметить, что синтаксис внутри квадратных скобок коренным образом отличается от остальной части регулярного выражения. К примеру, если вы поместите "." внутрь квадратных скобок, это позволит квадратным скобкам совпадать с "." буквально, также как 1 и 2 в примере выше. Для сравнения, "." помещенная вне квадратных скобок, будет интерпретирована как метасимвол, если не приставить "\". Мы можем получить выгоду из данного факта для вывода строк из /etc/fstab которые содержат строку dev.sda, как она записана:

$ **grep dev[.]sda /etc/fstab**

Также, мы могли бы набрать:

$ **grep "dev\.sda" /etc/fstab**

Эти регулярные выражения вероятно не удовлетворяют ни одной строчке из вашего */etc/fstab* файла.

### Матасимвол \*

Некоторые метасимволы сами по себе не соответствуют ничему, но изменяют значение предыдущего символа. Один из таких символов, это \* (звездочка), который используется для сопоставления нулевому или большему числу повторений предшествующего символа. Заметьте, это значит, что \* имеет другое значение в регулярках, нежели в глоббинге. Вот несколько примеров, и обратите особое внимание на те случаи где сопоставление регулярных выражений отличается от glob-подстановок:

* **ab\*c** совпадает с «abbbbc», но не с «abqc» (в случае glob-подстановки, обе строчки будут удовлетворять шаблону. Вы уже поняли почему?)
* **ab\*c** совпадает с «abc», но не с «abbqbbc» (опять же, при glob-подстановке, шаблон сопоставим с обоими строчками)
* **ab\*c** совпадает с «ac», но не с «cba» (в случае глоббинга, ни «ac», ни «cba» не удовлетворяют шаблону)
* **b[cq]\*e** совпадает с «bqe» и с «be» (glob-подстановке удовлетворяет «bqe», но не «be»)
* **b[cq]\*e** совпадает с «bccqqe», но не с «bccc» (при глоббинге шаблон точно так же совпадет с первым, но не со вторым)
* **b[cq]\*e** совпадает с «bqqcce», но не с «cqe» (так же и при glob-подстановке)
* **b[cq]\*e** удовлетворяет «bbbeee» (но не в случае глоббинга)
* **.\*** сопоставим с любой строкой (glob-подстановке удовлетворяют только строки начинающиеся с ".")
* **foo.\*** совпадет с любой подстрокой начинающийся с «foo» (в случае glob-подстановки этот шаблон будет совпадать со строками, начинающимися с четырех символов «foo.»)

Итак, повторим для закрепления: строчка «ac» подходит под регулярное выражение «ab\*c» потому, что звездочка также позволяет повторение предшествующего выражения (b) ноль раз. И опять, ценно отметить для себя, что метасимвол \* в регулярках интерпретируется совершенно иначе, нежели символ \* в glob-подстновках.

### Начало и конец строки

Последние метасимволы, что мы детально рассмотрим, это ^ и $, которые используются для сопостовления началу и концу строки, соответственно. Воспользовавшись ^ в начале вашего regex, вы «прикрепите» ваш шаблон к началу строки. В следующием примере, мы используем регулярное выражение ^#, которое удовлетворяет любой строке начинающийся с символа #:  
  
$ **grep ^# /etc/fstab**  
# /etc/fstab: static file system information.  
#

### Полнострочные регулярки

^ и $ можно комбинировать, для сопоставлений со всей строкой целиком. Например, нижеследующая регулярка будет соответсвовать строкам начинающимся с символа #, а заканчивающимся символом ".", при произвольном количестве символов между ними:

$ **grep '^#.\*\.$' /etc/fstab**  
# /etc/fstab: static file system information.

В примере выше мы заключили наше регулярное выражение в одиночные кавычки, чтобы предотвратить интерпретирование символа $ командной оболочкой. Без одиночных кавычек $ исчез бы из нашей регулярки еще даже до того, как grep мог его увидеть.

**24. Ввод и вывод символов и строк. Функции для работы со строками.**

Ввод и вывод информации осуществляется через функции стандартной библиотеки. Прототипы рассматриваемых функций находятся в файле stdio.h. Эта библиотека содержит функции

* printf() — для вывода информации
* scanf() — для ввода информации.

**Объявление строк**

Строка в языке Си представляет собой одномерный массив символов, последним элементом которой является символ конца строки – нуль. Способы:

1.Объявления массива символов (не забудьте добавить место для завершающего нуля):

➜ char s[40+1];

2.Присвоить строковой переменной начальное значение (при этом длину строки компилятор может вычислить сам):

➜ char s[] = "Пример инициализации строки";

3. Неявное указание, что используется массив. В левой части от знака присваивания указывается указатель на символ:

➜ char \*s="Второй вариант инициализации”;

**Ввод-вывод:**

Для работы со строками есть набор функций. Для ввода со стандартного устройства ввода (клавиатуры) чаще всего используются библиотечные функциями из модуля стандартного ввода-вывода: scanf и gets.

Для ввода строки с помощью функции scanf, используют формат «%s», причем обратите внимание на то, что перед идентификатором строки не используется знак адреса «&», так как одномерный массив уже представлен указателем на его начало:

*scanf("%s", s);*

Функция gets() считывает символы до тех пор, пока не достигнет символа перехода на новую строку. Функция принимает все символы вплоть до символа перевода строки, но не включает его. К концу строки добавляется завершающий ноль (‘\0’). Функция gets() помещает считанную с клавиатуры последовательность символов в параметр типа строка и возвращает указатель на эту строку (если операция завершилась успешно), или NULL (в случае ошибки).

Для вывода строк на стандартное устройство вывода (экран монитора) можно использовать две функции printf и puts. В функции printf в качестве формата передается «%s». Удобство использования этой функции заключается в том, что помимо строки можно сразу выводит данные других типов. Особенность функции puts заключается в том, что после вывода строки автоматически происходит переход на следующую строку.

25 Файлы и функции рабата с ними

***Функция*** — это самостоятельная единица программы, которая спроектирована для реализации конкретной подзадачи.

Функция является подпрограммой, которая может содержаться в основной программе, а может быть создана отдельно (в библиотеке). Каждая функция выполняет в программе определенные действия.

**Определение функции**

Каждая функция в языке Си должна быть определена, то есть должны быть указаны:

* тип возвращаемого значения;
* имя функции;
* информация о формальных аргументах;
* тело функции.

Разбиение программ на функции дает следующие преимущества:

* Функцию можно вызвать из различных мест программы, что позволяет избежать повторения программного кода.
* Одну и ту же функцию можно использовать в разных программах.
* Функции повышают уровень модульности программы и облегчают ее проектирование.
* Использование функций облегчает чтение и понимание программы и ускоряет поиск и исправление ошибок.

С точки зрения вызывающей программы функцию можно представить как некий «черный ящик», у которого есть несколько входов и один выход. С точки зрения вызывающей программы неважно, каким образом производится обработка информации внутри функции. Для корректного использования функции достаточно знать лишь ее сигнатуру.

**26 Динамические массивы. Особенности выделения и освобождения памяти для многомерных массивов.**

Для выделения памяти на куче в си используется функция ***malloc*** (memory allocation) из библиотеки stdlib.h

|  |
| --- |
| void \* malloc(size\_t size); |

Пример:

Одномерный:

Void main() {

Int size; // размер массива

Int \*p = NULL; // массив

Scanf("%d", &size); // задание размера массива

P = (int\*) malloc(size \* sizeof(int)); // выделение памяти для заданного кол-ва

… // действия с массивом

Free (p); // освобождение памяти

}

Двумерный:

Для динамического создания двумерного массива сначала необходимо создать массив указателей, после чего каждому из элементов этого массива присвоить адрес нового массива. Для удаления массива необходимо повторить операцию в обратном порядке - удалить сначала подмассивы, а потом и сам массив указателей.

Void main() {

Int i,;

Int \*\*p;

Int size\_col; //кол-во столбцов

Int size\_row; //кол-во строк

Scanf("%d", &size\_col);

Scanf("%d", &size\_row);

P = (int\*\*) malloc(size\_row \* sizeof(int\*)); //выделение памяти под массив указателей

For (i = 0; i<size\_row; i++){

P[i] = (int\*) malloc(size\_col \* sizeof(int));}

…

For (i = 0; i<size\_row; i++){ //освобождение памяти

Free(p[i]);}

Free(p);

}

**27. Динамические объекты. Способы выделения и освобождения памяти.**

Для работы с динамической памятью в языке С используются следующие функции: malloc, calloc, free, realloc. Рассмотрим их подробнее.

Выделение (захват памяти) : void \*malloc(size\_t size);

В качестве входного параметра функция принимает размер памяти, которую требуется выделить. Возвращаемым значением является указатель на выделенный в куче участок памяти. Если ОС не смогла выделить память (например, памяти не хватило), то malloc возвращает 0.

После окончания работы с выделенной динамически памятью нужно освободить ее. Для этой цели используется функция free, которая возвращает память под управление ОС : void free(void \*ptr);

Если динамическая памятьне освобождена до окончания программы, то она освобождается автоматически при завершении программы. Тем не менее, явное освобождение ставшей ненужной памяти является признаком хорошего стиля программирования.

Пример: // выделения памяти под 1 000 элементов типа int

int \* p = (int \*) malloc(1000\*sizeof(int));

if (p==NULL) cout<< "\n память не выделена";

free (p); // освобождение дин. памяти

}

Указатель определен как локальный объект статической памяти. Динамическая память, выделенная однократно в блоке, доступна через указатель при каждом повторном входе в блок. Память нужно освободить только по окончании ее использования.

void f1 (int n)

{static int\* p = (int \*) calloc(n, sizeof(int));

if (n) // n !=0

p= (int \*) calloc(n, sizeof(int));

else

free (p);

. . .

}

void main ()

{. . .

f(50); //выделение дин. памяти с последующим освобождением

f1(100); //выделение дин. памяти (первое обращение)

. . .

f1(100); //работа с дин. памятью

f1 (0); // освобождение дин. памяти

}

Указатель является глобальным объектом по отношению к блоку. Динамическая память доступна во всех блоках, где "виден" указатель. Память нужно освободить только по окончании ее использования

#define n 100

int\* pG; //рабочий указатель для дин. памяти (глобальная переменная)

void init (int size)

{ int i;

for (i=0; i< size; i++) //цикл ввода чисел

{ printf("x[%d]=",i);

scanf("%d", &pG[i]);

}

int sum (int size)

{ int i,s=0;

for (i=0; i< size; i++) //цикл суммирования

s=+pG[i]);

return s;

}

void main ()

{

pG= (int \*) calloc(n, sizeof(int)); // выделение памяти

init (n); //работа с дин.памятью

printf(\ns=%d\n”,sum(n));

NULL; // освобождение памяти

}

### Арифметические операторы

|  |  |  |  |
| --- | --- | --- | --- |
| **Операция (выражение)** | | **Оператор** | **Синтаксис выражения** |
|
| [Присваивание](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D1%8F_%D0%BF%D1%80%D0%B8%D1%81%D0%B2%D0%B0%D0%B8%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F_%D0%B2_%D0%A1%2B%2B) | | **=** | a **=** b |
| Сложение | | **+** | a **+** b |
| Вычитание | | **-** | a **-** b |
| Унарный плюс | | **+** | **+**a |
| Унарный минус | | **-** | **-**a |
| Умножение | | **\*** | a **\*** b |
| Деление | | **/** | a **/** b |
| Операция модуль ([остаток от деления](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D1%81_%D0%BE%D1%81%D1%82%D0%B0%D1%82%D0%BA%D0%BE%D0%BC) целых чисел)[[note 1]](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80%D1%8B_%D0%B2_C_%D0%B8_C%2B%2B#cite_note-modulo-1) | | **%** | a **%** b |
| [Инкремент](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%BA%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D1%82) | префиксный | **++** | **++**a |
| суффиксный (постфиксный) | **++** | a**++** |
|
| [Декремент](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%BA%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D1%82) | префиксный | **--** | **--**a |
| суффиксный (постфиксный) | **--** | a**--** |
|

### Операторы сравнения

|  |  |  |
| --- | --- | --- |
| **Операция (выражение)** | **Оператор** | **Синтаксис выражения** |
|
| Равенство | **==** | a **==** b |
| Неравенство | **!=** | a **!=** b |
| Больше | **>** | a **>** b |
| Меньше | **<** | a **<** b |
| Больше или равно | **>=** | a **>=** b |
| Меньше или равно | **<=** | a **<=** b |

### Логические операторы

|  |  |  |
| --- | --- | --- |
| **Операция (выражение)** | **Оператор** | **Синтаксис выражения** |
|
| Логическое отрицание, НЕ | **!** | **!**a |
| Логическое умножение, И | **&&** | a **&&** b |
| Логическое сложение, ИЛИ | **||** | a **||** b |

### Побитовые операторы

|  |  |  |
| --- | --- | --- |
| **Операция (выражение)** | **Оператор** | **Синтаксис выражения** |
|
| Побитовая инверсия | **~** | **~**a |
| Побитовое И | **&** | a **&** b |
| Побитовое ИЛИ (or) | **|** | a **|** b |
| Побитовое исключающее ИЛИ (xor) | **^** | a **^** b |
| Побитовый сдвиг влево[[note 3]](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80%D1%8B_%D0%B2_C_%D0%B8_C%2B%2B#cite_note-bitshift-3) | **<<** | a **<<** b |
| Побитовый сдвиг вправо[[note 3]](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80%D1%8B_%D0%B2_C_%D0%B8_C%2B%2B#cite_note-bitshift-3)[[note 4]](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80%D1%8B_%D0%B2_C_%D0%B8_C%2B%2B#cite_note-rightbitshift-4) | **>>** | a **>>** b |

### Составное присваивание

|  |  |  |  |
| --- | --- | --- | --- |
| **Операция (выражение)** | **Оператор** | **Синтаксис выражения** | **Значение** |
|
| Сложение, совмещённое с присваиванием | **+=** | a **+=** b | a **=** a **+** b |
| Вычитание, совмещённое с присваиванием | **-=** | a **-=** b | a **=** a **-** b |
| Умножение, совмещённое с присваиванием | **\*=** | a **\*=** b | a **=** a **\*** b |
| Деление, совмещённое с присваиванием | **/=** | a **/=** b | a **=** a **/** b |
| Вычисление остатка от деления, совмещённое с присваиванием[[note 1]](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80%D1%8B_%D0%B2_C_%D0%B8_C%2B%2B#cite_note-modulo-1) | **%=** | a **%=** b | a **=** a **%** b |
| Побитовое «И» (AND), совмещённое с присваиванием | **&=** | a **&=** b | a **=** a **&** b |
| Побитовое «ИЛИ» (or), совмещённое с присваиванием | **|=** | a **|=** b | a **=** a **|** b |
| Побитовое «исключающее ИЛИ» (xor), совмещённое с присваиванием | **^=** | a **^=** b | a **=** a **^** b |
| Побитовый сдвиг влево, совмещённый с присваиванием | **<<=** | a **<<=** b | a **=** a **<<** b |
| Побитовый сдвиг вправо, совмещённый с присваиванием[[note 4]](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80%D1%8B_%D0%B2_C_%D0%B8_C%2B%2B#cite_note-rightbitshift-4) | **>>=** | a **>>=** b | a **=** a **>>** b |

### Операторы работы с указателями и членами класса

|  |  |
| --- | --- |
| **Оператор** | **Синтаксис** |
|
| Обращение к элементу массива | a**[**b**]** |
| Непрямое обращение («объект, на который указывает *a*») | **\***a |
| Ссылка («адрес *a*») | **&**a |
| Обращение к члену структуры («член *b* объекта, на который указывает *a*») | a**->**b |
| Обращение к члену структуры («член *b* объекта *a*») | a**.**b |
| Член, на который указывает *b* в объекте, на который указывает *a*[[note 6]](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80%D1%8B_%D0%B2_C_%D0%B8_C%2B%2B#cite_note-arrowstar-6) | a**->\***b |
| Член, на который указывает *b* в объекте *a* | a**.\***b |

### Другие операторы[[править](https://ru.wikipedia.org/w/index.php?title=%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80%D1%8B_%D0%B2_C_%D0%B8_C%2B%2B&veaction=edit&section=8) | [править код](https://ru.wikipedia.org/w/index.php?title=%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80%D1%8B_%D0%B2_C_%D0%B8_C%2B%2B&action=edit&section=8)]

|  |  |
| --- | --- |
| **Оператор** | **Синтаксис** |
|
| [Функтор](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82) | a**(**a1, a2**)** |
| Оператор «запятая» | a**,** b |
| [Тернарная условная операция](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%80%D0%BD%D0%B0%D1%80%D0%BD%D0%B0%D1%8F_%D1%83%D1%81%D0%BB%D0%BE%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D1%8F) | a **?** b **:** c |
| Оператор расширения области видимости | a**::**b |
| [Sizeof](https://ru.wikipedia.org/w/index.php?title=Sizeof&action=edit&redlink=1) (размер) | **sizeof**(a)[[note 7]](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80%D1%8B_%D0%B2_C_%D0%B8_C%2B%2B#cite_note-sizeof-7) **sizeof**(*type*) |
| [Align-of](https://ru.wikipedia.org/w/index.php?title=Alignof&action=edit&redlink=1) ([выравнивание](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%80%D0%B0%D0%B2%D0%BD%D0%B8%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)) | **alignof**(*type*) или **\_Alignof**(*type*)[[note 8]](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80%D1%8B_%D0%B2_C_%D0%B8_C%2B%2B#cite_note-alignof-8) |
| [Интроспекция](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D1%80%D0%BE%D1%81%D0%BF%D0%B5%D0%BA%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) | **typeid**(a) **typeid**(*type*) |
| [Приведение типа](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5_%D1%82%D0%B8%D0%BF%D0%B0) | (*type*) a |
|
| Выделение памяти | **new** *type* |
| Выделение памяти для массива | **new** *type***[**n**]** |
| Освобождение памяти | **delete** a |
| Освобождение памяти, занятой массивом | **delete[]** a |

30 см. вопрос 22

30

**СТАНДАРТНАЯ БИБЛИОТЕКА, КОЛЛЕКЦИИ ФУНКЦИЙ**(ЧЁ БЛЯ?)

**Стандартной библиотекой языка Си** (также известная как **libc**, **crt**) называется часть стандарта [ANSI C](https://ru.wikipedia.org/wiki/ANSI_C), посвященная [заголовочным файлам](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B3%D0%BE%D0%BB%D0%BE%D0%B2%D0%BE%D1%87%D0%BD%D1%8B%D0%B9_%D1%84%D0%B0%D0%B9%D0%BB) и [библиотечным подпрограммам](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)). Является описанием реализации общих операций, таких как обработка [ввода-вывода](https://ru.wikipedia.org/wiki/%D0%92%D0%B2%D0%BE%D0%B4-%D0%B2%D1%8B%D0%B2%D0%BE%D0%B4) и [строк](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D0%BE%D0%BA%D0%BE%D0%B2%D1%8B%D0%B9_%D1%82%D0%B8%D0%BF), в [языке программирования Си](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F_%D0%A1%D0%B8). Стандартная библиотека языка Си — это описание [программного интерфейса](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D1%8B%D0%B9_%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81), а не настоящая библиотека, пригодная для использования в процессе [компиляции](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).

Имя и характеристики каждой функции указываются в файле, именуемом [заголовочным файлом](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B3%D0%BE%D0%BB%D0%BE%D0%B2%D0%BE%D1%87%D0%BD%D1%8B%D0%B9_%D1%84%D0%B0%D0%B9%D0%BB), но текущая реализация функций описана отдельно в [библиотечном файле](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)). Наименование и возможности заголовочных файлов становятся общими, но организация библиотек по-прежнему остается разнотипной. Стандартная библиотека обычно поставляется вместе с [компилятором](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80). Так как компиляторы языка Си часто обеспечивают расширенную функциональность, не определенную стандартом [ANSI C](https://ru.wikipedia.org/wiki/ANSI_C), стандартная библиотека одного компилятора несовместима со стандартными библиотеками других компиляторов.

**31. Рекурсивные функции.**

В языке Си функции могут *вызывать сами себя* непосредственно или косвенно, т.е. могут быть *рекурсивными*. Если функция непосредственно вызывает саму себя – то это называется *прямой рекурсией*, а если функция вызывает какую-либо другую функцию, которая либо сама, либо посредством другой функции вызывает исходную функцию – то это называется *косвенной рекурсией*.

Каждая цепочка рекурсивных вызовов должна на каком-то шаге завершиться. Условие полного окончания работы рекурсивной функции должно находиться в самой функции (иначе произойдет зацикливание), а именно, любая рекурсивная функция должна содержать рекурсивный вызов внутри условного оператора.

Применять рекурсивные методы программирования стоит в тех задачах, где рекурсия использована в определении обрабатываемых данных. Это, как правило, связано с такими динамическими структурами данных как стеки, деревья, очереди и др. Многие задачи, решаемые при помощи рекурсии, более эффективно решаются либо с помощью итеративных алгоритмов либо с помощью подпрограмм. Например, вычисление факториала, которое мы рассмотрим, удобно для объяснения рекурсии, однако не дает никакого выигрыша в программной реализации. Более того, рекурсивный алгоритм вычисления факториала работает медленнее итеративного алгоритма, за счет накладных расходов на вызов функции и возврат значений.

Рассмотрим пример программы вычисляющей факториал положительного числа.

|  |
| --- |
| #include <stdio.h>  #include <conio.h>  #include <iostream>  using namespace std;  void main()  {  int f(int k); //функция вычисления факториала  int fact = f(5);  cout << "fact! = " << fact;  getch();  }  // Рекурсивная функция вычисления факториала  int f(int k)  {  if (k < 0) return 0;  if (k == 0) return 1; //т.к. 0! == 1  return k \* f(k-1);  } |

При первом вызове **f(5)** функция возвращает выражение **5\*f(4)**, т.е. функция **f()** фактически не возвращает значение, а вызывает сама себя с другим значением. Рекурсивные вызовы будут продолжаться до тех пор, пока **k** не станет равным **0**. Будет создана следующая цепочка возвращаемых выражений:

**5\*f(4), 4\*f(3), 3\*f(2), 2\*f(1), 1\*f(0)**

Вызов **f(0)**не провоцирует дальнейших вызовов, а возвращает значение 1, произведение **1\*1** будет возвращено предыдущему вызову и т.д. до вызова **f(5)**, которому возвращается значение **120**. Тем самым будут организованы следующие умножения:

**1\*1\*2\*3\*4\*5**, а в общем случае **1\*1\*2\*3\*4\*5\*…\*(k-1)\*k**

**32. Типы переменных, область видимости и время жизни, дополнительные спецификаторы**

Типы переменных:

- локальные (внутри функции(блока))

* глобальные (вне функции(блока))

**Время жизни определяется по следующим правилам:**

1. Переменная, объявленная глобально (т.е. вне всех блоков), существует на протяжении всего времени выполнения программы.

2. Локальные переменные (т.е. объявленные внутри блока) имеют время жизни только на период выполнения того блока, в котором они объявлены.

**Видимость переменных и функций в программе определяется следующими правилами:**

1. Переменная, объявленная или определенная глобально, видима от точки объявления или определения до конца исходного файла. Можно сделать переменную видимой и в других исходных файлах, для чего в этих файлах следует ее объявить с классом памяти extern.

2. Переменная, объявленная или определенная локально, видима от точки объявления или определения до конца текущего блока. Такая переменная называется локальной.

3. Переменные из объемлющих блоков, включая переменные объявленные на глобальном уровне, видимы во внутренних блоках. Эту видимость называют вложенной. Если переменная, объявленная внутри блока, имеет то же имя, что и переменная, объявленная в объемлющем блоке, то это разные переменные, и переменная из объемлющего блока во внутреннем блоке будет невидимой.

4. Функции с классом памяти static видимы только в исходном файле, в котором они определены. Всякие другие функции видимы во всей программе.

Метки в функциях видимы на протяжении всей функции.

Имена формальных параметров, объявленные в списке параметров прототипа функции, видимы только от точки объявления параметра до конца объявления функции.

**Дополнительные спецификаторы (??????)**

Спецификатор точности следует за спецификатором минимума ширины поля (если он есть). Он состоит из точки, за которой следует целое число. Его точное значение зависит от типа данных, к которым он применяется.

При применении спецификатора точности к вещественным данным, используя спецификаторы %f, %е или %Е, он определяет число выводимых цифр дробной части. Например, %10.4f выводит числа, по крайней мере, с длиной строки в десять символов, причем под дробную часть отводится четыре символа. Если не указать точность, то по умолчанию используется шесть символов. Когда спецификатор точности применяется к %g или %G, он определяет число значимых цифр. Применение спецификатора точности к строкам ограничивает длину строки. Например, %5.7s выводит строку, по крайней мере, с пятью, но не более чем с семью символами. Если строка длиннее, чем максимальная указанная ширина, оставшиеся символы отбрасываются.

Когда спецификатор точности применяется к целым типам, он указывает минимальное число цифр, отводимых под каждое число. Для достижения требуемой длины к числу добавляются нули. Следующая программа иллюстрирует применение спецификатора точности:

#include <stdio.h>

int main (void)

{

printf ("%. 4f\n\",123.1234567);

printf("%3.8d\n", 1000);

printf("%10.15 s\n", "This is a simple test.");

return 0;

}

В результате ее работы получаем:

123.1235

00001000

This is a simple test.

**33. Функции с переменным числом параметров**

По умолчанию параметры передаются функции через стек. Поэтому, технически, нет ограничения на количество передаваемых параметров – “запихать” можно сколько угодно. Проблема в том, как потом функция будет разбирать переданные параметры. Функции с переменным числом параметров объявляются как обычные функции, но вместо недостающих аргументов ставится многоточие. Пусть мы хотим сделать функцию, которая складывает переданные ей числа, чисел может быть произвольное количество. Необходимо каким-то образом передать функции число параметров. Во-первых, можно явно передать число параметров обязательным аргументом. Во-вторых, последний аргумент может иметь некоторое «терминальное» значение, наткнувшись на которое функция закончит выполнение.

Общий принцип работы следующий: внутри функции берём указатель на аргумент, далее двигаемся к следующему аргументу, увеличивая значение указателя.

OLD SCHOOL

Делаем всё вручную. Функция, которая складывает переданные ей аргументы

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | #include <stdio.h>  #include <conio.h>  #include <stdlib.h>    #define UNSIGNED\_OVERFLOW -4    **unsigned** summ(**unsigned** **char** num, **unsigned** first, ...) {  **unsigned** sum = 0;  **unsigned** testsum = 0;  **unsigned** \*p = &first;  **while** (num--) {          testsum += \*p;  **if** (testsum >= sum) {              sum = testsum;          } **else** {  **exit**(UNSIGNED\_OVERFLOW);          }          p++;      }  **return** sum;  }    **void** main() {  **int** sum = summ(5, 1u, 2u, 3u, 4u, 5u);  **printf**("summ = %u\n", sum);      sum = summ(7, 0u, 27u, 0u, 4u, 5u, 60u, 33u);  **printf**("summ = %u\n", sum);      getch();  } |

Первый параметр – число аргументов. Это обязательный параметр. Второй аргумент – это первое переданное число, это тоже обязательный параметр. Получаем указатель на первое число

|  |  |
| --- | --- |
| 10 | **unsigned** \*p = &first; |

Далее считываем все числа и складываем их. В этой функции мы также при сложении проверяем на переполнение типа unsigned.