1. Указатели. Инициализация указателей. Операции с указателями.

***Указатель*** — переменная, содержащая адрес объекта. Указатель не несет информации о содержимом объекта, а содержит сведения о том, где размещен объект. Указатели часто используются при работе с массивами. Указатель — это тоже переменная, которая размещается в памяти. Она тоже имеет адрес, а ее значение является адресом некоторой другой переменной.

Для работы с указателями в Си определены две операции:

* операция \* (звездочка) — позволяет получить значение объекта по его адресу;
* операция & (амперсанд) — позволяет определить адрес переменной.

**Инициализация:**

* ***Первый способ*:** описать переменные программы.

**float r, b;**  
**float \*ukr = &r, \*ukb = &b;**

* ***Второй способ*:** присвоить указателю значение другого указателя, к этому моменту уже правильно инициализированного, например:

**char \*uk\_simv\_rus, \*uk\_simv\_lat;**  
**char simvsmoll, simvlarge;**  
**. . .**  
**uk\_simv\_lat = &simvsmoll;**  
**uk\_simv\_rus = uk\_simv\_lat;**

* ***Третий способ*:** использовать одну из встроенных функции распределения памяти, например, malloc. Функция возвращает адрес выделенной памяти в виде константы-указателя типа **void**.

**int \*x = (int \*)malloc(sizeof(int));**

**Операции.** Над указателями можно выполнять унарные операции: инкремент и декремент.

В бинарных операциях сложения и вычитания могут участвовать указатель и величина типа int. При этом результатом операции будет указатель на исходный тип, а его значение будет на указанное число элементов больше или меньше исходного.

int \*ptr1, \*ptr2, a[10];

int i=2;

ptr1=a(i+4); /\* равно адресу элемента a[6] \*/

ptr2=ptr1-i; /\* равно адресу элемента a[4] \*/

В операции вычитания могут участвовать два указателя на один и тот же тип. Результат такой операции имеет тип int и равен числу элементов исходного типа между уменьшаемым и вычитаемым, причем если первый адрес младше, то результат имеет отрицательное значение.

int \*ptr1, \*ptr2, a[10];

int i;

ptr1=a+4;

ptr2=a+9;

i=ptr1-ptr2; /\* равно 5 \*/

i=ptr2-ptr1; /\* равно -5 \*/

Значения двух указателей на одинаковые типы можно сравнивать в операциях ==, !=, <, <=",">, >= при этом значения указателей рассматриваются просто как целые числа, а результат сравнения равен 0 (ложь) или 1 (истина).

Чаще всего указатели используют:

1.Для работы с массивом  
2.Передачи данных в функцию

3.Построение сложных динамических структур

1. Указатели в C++. Операция разыменования. Константные указатели и указатели на константы. Ссылки в C++.

**Разыменованием указателя**называется конструкция, позволяющая получить доступ к **значению**переменной, на которую ссылается указатель (т. е. адрес которой хранит указатель). Разыменование указателя осуществляется с использованием символа «\*» («звёздочка») перед именем указателя.

**Константный указатель** — это когда адрес, хранимый внутри указательной переменной, изменять не положено.

**Указатель на константу** — это когда значение, вытаскиваемое из адреса, воздействием указательной переменной изменять не положено.

**Ссылка** – константный указатель (объект, указывающий на положение другой переменной). Отличается от указателя тем, что не требует разыменования. Над ссылками невозможны арифметические операции. Формат объявления ссылки: тип & имя = инициализатор;

Инициализатор – то, на что указывает ссылка.

**Запомните следующие правила:**

* Переменная-ссылка должна явно инициализироваться при ее описании, кроме случаев, когда она является параметром функции, описана как extern или ссылается на поле данных класса.
* После инициализации ссылке не может быть присвоена другая переменная.
* Тип ссылки должен совпадать с типом величины, на которую она ссылается.
* Не разрешается определять указатели на ссылки, создавать массивы ссылок и ссылки на ссылки.
* Ссылки применяются чаще всего в качестве параметров функций и типов возвращаемых функциями значений.
* Ссылка, в отличие от указателя, не занимает дополнительного пространства в памяти и является просто другим именем величины. Операция над ссылкой приводит к изменению величины, на которую она ссылается.

1. Массивы. Динамические массивы. Динамические многомерные массивы.

**Массив** — это непрерывный участок памяти, содержащий последовательность объектов одинакового типа, обозначаемый одним именем.

**Динамическим массивом** называют *массив* с переменным размером, то есть количество элементов может изменяться во *время выполнения* программы. Для создания двумерного *динамического массива* вначале нужно распределить *память* для массива указателей на одномерные массивы, а затем выделить *память* для одномерных массивов. При *динамическом распределении памяти* для массивов следует описать соответствующий *указатель*, которому будет присвоено *значение* адреса начала области выделенной памяти.

Операции new и delete предназначены для динамического распределения памяти компьютера. Операция new выделяет память из области свободной памяти, а операция delete высвобождает выделенную память. Если память невозможно выделить, например, в случае отсутствия свободных участков, то возвращается нулевой указатель, то есть указатель вернет значение 0. Выделение памяти возможно под любой тип данных.

Многомерный массив объявляется следующим образом:  
тип имя [разнерN] ... [размер2] [размер1];

Массивы, имеющие более трех размерностей, используются чрезвычайно редко, поскольку требуется большой объем памяти для их хранения.

Важно знать, что в многомерных массивах требуется некоторое время на вычисление каждого индекса. Это означает, что доступ к элементу в многомерных массивах происходит медленнее, чем доступ в одномерных массивах. По этой и другим причинам, если возникает необходимость в многомерных массивах, для них чаще всего память выделяется динамически.

1. Строки. Объявление и инициализация.

Строка в СИ – это массив элементов типа char, в конце которого помещен символ '\0' (нуль-терминатор), компилятор сам добавит его в конец строки. Символьные строки состоят из набора символьных констант, заключённых в двойные кавычки. При объявлении строкового массива необходимо учитывать наличие в конце строки нуль-терминатора, и отводить дополнительный байт под него. Строка при объявлении может быть инициализирована начальным значением.

При объявлении строки не обязательно указывать её размер, но при этом обязательно нужно её инициализировать начальным значением. Тогда размер строки определится автоматически. В СИ строки заключаются в двойные кавычки. Имя строки является константным указателем на первый символ.

В языке Си отдельного типа данных «строки символов» нет. Работа со строками реализована путем использования одномерных массивов типа char, т.е. строка символов – это одномерный массив типа char, заканчивающийся нулевым байтом.

Строковая константа – это набор символов, заключенных в двойные кавычки.

1. Функции для работы со строками и символами.

**Действия над строками:**

* инициализация;
* ввод/вывод (gets, puts);
* копирование строк (strcpy, strncpy);
* конкатенация или объединение (strcat, strncat);
* определение длины строки (strlen);
* сравнение строк (strcmp, strncmp);
* поиск символов в строке (strchr, strcspn, strspn, strprbk).

**Вывод строк**

* использовать функцию **printf()** со спецификатором **%s**
* использовать функцию **puts()** int puts(const char \*s);
* использовать функцию **fputs()**, указав в качестве второго параметра стандартный поток для вывода **stdout**.

Функция **puts()** переносит вывод на следующую строку, а функция **fputs()** не переносит.

**Ввод строк**

* Функция **gets** приостанавливает работу программы, читает строку символов, введенных с клавиатуры, и помещает в символьный массив, имя которого передаётся функции в качестве параметра. Завершением работы функции **gets()** будет являться символ, соответствующий клавише ввод и записываемый в строку как нулевой символ. Это чревато тем, что мы можем выйти за рамки массива.

char\* gets(char \*str);

* Функция **fgets()** принимает на вход три аргумента: переменную для записи строки, размер записываемой строки и имя потока, откуда взять данные для записи в строку, в данном случае — **stdin**. Она не дает случиться переполнению. Автоматически откинет символы, вышедшие за пределы массива строки. При использовании необходимо очищать буфер ввода.
* **scanf()**: ввод с форматированием данных

Прототипы функций gets и puts описаны в заголовочном файле stdio.h.

int **strlen**(char \*S)

int ****strcpy****(char \*S1, char \*S2)

**strcat**(char \*S1, char \*S2)

int **strcmp**(char \*S1, char \*S2)

Функции преобразования строки S в число:

- целое: int **atoi**(char \*S);

- длинное целое: long **atol**(char \*S);

- действительное: double **atof**(char \*S);

при ошибке данные функции возвращают значение 0.

Функции преобразования числа V в строку S:

- целое: **itoa**(int V, char \*S, int kod);

- длинное целое: **ltoa**(long V, char \*S, int kod);

**del\_c()**, в которой удаляется символ "с" из строки s каждый раз, когда он встречается.

void del\_c( char s[ ], int c)

1. Структуры. Объявление. Доступ к полям.

Структура — это агрегатный тип данных, так как может содержать в себе разнотипные элементы. в структуре столько полей, сколько вам угодно и они могут иметь разные типы. Объявление структуры приводит к образованию шаблона, который используется для создания объектов структуры.

**Формат объявления** структуры выглядит так:

struct ярлык

{

    тип имя\_переменной;

    ...

    тип имя\_переменной;

} структурные переменные;

Ключевое слово struct объявляет компилятору о структуре. Имя структуры идентифицирует структуру и является спецификатором типа. Ярлык – имя типа структуры. Структурные переменные – разделенный запятыми список имен переменных. Или ярлык, или структурные переменные могут отсутствовать, но не оба.

Для доступа к полям структуры используется точка:

Имя\_структуры.имя\_поля

1. Структуры. Массивы структур. Массивы и структуры в структурах.

Из структур можно создавать массивы также, как массивы других типов. И все форматы определения массива структур будут аналогичны определению массивов других типов:

struct person people[10];

В данном случае определен массив структур person из 10 элементов.

Обращение к элементам массива структур происходит по индексу people[0]. А чтобы обратиться к элементу структуры из массива, после индекса указывается имя элемента структуры: people[i].name

И также как с массивами других типов с массивами структур можно использовать указатели.

Поле структуры может быть простым и составным. Простое поле –обычный базовый тип данных. Составные типы – одномерные и многомерные массивы различных типов данных и структур.

чтобы указать определенную структуру, находящуюся в массиве структур, необходимо указать имя этого массива с определенным индексом. А если нужно указать индекс определенного элемента в структуре, то необходимо указать индекс этого элемента. Таким образом, в результате выполнения следующего выражения первому символу члена name, находящегося в третьей структуре из addr\_list, присваивается значение 'X'. addr\_list[2].name[0] = 'X';

Когда структура является членом другой структуры, то она называется вложенной структурой. Например, ниже структура addr вложена в emp:  
struct emp {  
struct addr address;  
float wage;  
} worker;

worker.wage = 35000.00;  
worker.address.zip = 98765;

1. Передача структур в функции

Очень часто требуется писать функции, которые принимают структуры в качестве аргумента или возвращают структуру.

Существует 2 способа передачи структуры в функцию в качестве параметра:

* передача структуры по значению. При такой передаче делается копия структурной переменной в памяти. Если структура имеет большой размер, то такой способ неэффективен. Преимуществом этого способа есть то, что все манипуляции с копией структуры в функции не влияют на исходную переменную;
* передача указателя на структуру. В этом случае передается только указатель на структуру а не вся структура. Если структура занимает большой объем памяти, то такой способ обеспечивает быструю передачу значений структурной переменной в функцию. Недостатком этого способа есть то, что в функции случайно можно изменить значения исходной структурной переменной, в тех случаях когда это нежелательно.

существуют 2 способа возврата:

* возврат структуры по значению;
* возврат указателя на структуру.

Также можно передать только какой-то один член структуры.

1. Указатели на структуры

struct bal {  
float balance;  
char name[80];  
} person;

Для получения адреса структурной переменной следует поместить оператор & перед именем структуры.

struct bal \*p; /\* объявление указателя на структуру \*/

p = &регson; //помещает адрес структуры person в указатель р.

Указатели на структуры можно создавать и для безымянных структурных типов:

struct

{

    int age;

    char name[20];

} \*p1, \*p2;

В качестве значения такому указателю присваивается адрес объекта структуры того же типа:

struct person kate = {31, "Kate"};

struct person \*p\_kate = &kate;

Используя указатель на структуру, можно получить доступ к ее элементам. Для этого можно воспользоваться двумя способами. Первый способ представляет применение операции разыменования:

(\*указатель\_на\_структуру).имя\_элемента

Второй способ предполагает использование операции **->** (операция стрелка):

указатель\_на\_структуру->имя\_элемента

\*Второй способ записи удобнее и шире используется

1. Битовые поля.

**Битовые поля** - особый тип структуры, определяющий какую длину имеет каждое поле.

Битовые поля полезны по нескольким причинам:

1. Если ограничено место для хранения информации, можно сохранить несколько логических переменных в одном байте.

2. Некоторые интерфейсы устройств передают информацию, закодировав биты в один байт.

3. Некоторым процедурам кодирования необходимо получить доступ к отдельным битам в байте.

4. Вносят ясность в программу.

В языках C и C++ при объявлении битового поля используется символ двоеточия (:). После двоеточия указывается константное выражение, определяющее количество битов в битовом поле. Стандартный вид объявления битовых полей следующий:

struct имя структуры {

тип имя1: длина;

тип имя2: длина;

...

тип имяN: длина;

}

Битовые поля должны объявляться как int, unsigned или signed.

Битовые поля имеют некоторые ограничения.

* 1. Нельзя получить адрес переменной битового поля, так как битовое поле может находиться внутри байта.
  2. Переменные битового поля не могут помещаться в массив.
  3. Переходя с компьютера на компьютер нельзя быть уверенным в порядке изменения битов (слева направо или справа налево). Любой код, использующий битовые поля, зависит от компьютера.

1. Объединения. Объявление. Доступ к полям.

**Объединение** - это объект, позволяющий нескольким переменным различных типов занимать один и тот же участок памяти.

Определяются объединения с помощью служебного слова union.

Можно объявлять переменную, поместив её имя в конец определения или используя отдельный оператор объявления.

union un\_t {

long L;

unsigned i1;

int i2;

char c[4];

} *UNI*;

**Или**

Union un\_t *UNI;*

Размер памяти, выделяемый объединению, равен максимальному из размеров памяти, выделяемых для отдельных полей шаблона. Тип поля может быть любым, в том числе и структурой. Структуры могут быть членами объединения, и объединение может быть членом структуры. Разрешается создание и массива объединений.

Доступ к полям объединения выполняется, аналогично структурам, через уточненное имя:

имя\_объединения.имя\_поля

(\*указатель\_на\_объединение).имя\_поля

указатель\_на\_объединение->имя\_поля

ссылка\_на\_объединение.имя\_поля

Основное достоинство объединений – возможность разных трактовок одного и того же содержимого памяти. Объединения применяют для экономии памяти в случае, когда объединяемые элементы логически существуют в разные моменты времени либо требуется разнотипная интерпретация поля данных.

Объявление объединения мб анонимным. Если объявление анонимно, а имена полей уникальны, то их можно использовать как переменные.

union {

long L;

unsigned i1;

int i2;

char c[4];

} UNI;

1. Перечисления. Объявление. Доступ к полям.

**Перечисления** - набор целочисленных констант, определяющий все допустимые значения, которые сожет принимать переменная. Определяются с помощью ключевого слова enum, указывает на начало перечисляемого типа.

Определение перечисляемого типа данных имеет вид *enum ярлык {список\_имён} список\_переменных*;

Переменным перечислимого типа (список\_переменных) можно присваивать только именованные значения перечислимых констант (список\_имён).

В перечислениях каждому символу соответствуют целочисленные значения и их можно использовать в любом целочисленном выражении. Можно определить один или несколько символ, следующие за ним будут больше на 1.

    enum COLOR{

        ORANGE,

        YELLOW,

        GREEN,

        LIGHT\_BLUE,

        BLUE,

        VIOLETTE

    };

 int main(){

COLOR color; //color - объект данного перечисления

color = YELLOW; //доступ к полю

1. Классы памяти переменных

Класс памяти определяет время жизни и область видимости. Если класс памяти не указан, то он определяется по умолчанию из контекста объявления.

Есть 3 основных места, где объявляется переменная:

1. Внутри функции (локальная)
2. При определении параметров (формальная)
3. Вне функции (глобальная)

Объекты классов и классы могут объявляться с такими классами памяти:

* автоматический (auto) – локальные переменные, “время жизни” ограничено временем выполнения ф-ии или блока, где она определена;
* регистровый (register) – выделение для хранения данных во внутренних регистрах процессора;
* внешний (extern) – для передачи между модулями;
* статический (static). – в фиксированной области памяти (один модуль)
* изменчивый (mutable) – используется с константными функциями.

1. Область действия переменных.

В одномодульной программе без вложенных функций и без использования ООП может существовать только два типа области видимости: глобальная и локальная. Прочие типы существуют только при наличии в языке определённых синтаксических механизмов.

* **Глобальная область видимости** — идентификатор доступен во всём тексте.
* **Локальная область видимости** — идентификатор доступен только внутри определённой функции.
* **Видимость в пределах модуля** может существовать в [модульных программах](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)), состоящих из нескольких отдельных фрагментов кода, обычно находящихся в разных файлах.
* [**Пакет**](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D0%BA%D0%B5%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))**или**[**пространство имён**](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D1%81%D1%82%D0%B2%D0%BE_%D0%B8%D0%BC%D1%91%D0%BD_(%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))**.** В глобальной области видимости искусственно выделяется поименованная подобласть. Имя «привязывается» к этой части программы и существует только внутри неё. Вне данной области имя либо вообще недоступно, либо доступно ограниченно.

В [ООП](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%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)-языках дополнительно к вышеперечисленным могут поддерживаться специальные ограничения области видимости, действующие только для членов классов:

* **Приватная (личная, закрытая)** ([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) *private*) - имя доступно только внутри методов своего [класса](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_(%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)).
* **Защищённая** ([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) *protected*) - имя доступно только внутри своего класса и его классов-потомков.
* **Общая** ([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) *public*) - имя доступно в пределах области видимости, к которой относится его класс.

1. Модульное программирование.

Модульное программирование — это организация программы как совокупности небольших независимых блоков, называемых модулями. Модуль — функционально законченный фрагмент программы, оформленный в виде отдельного файла с исходным кодом.

Удобство использования модульной архитектуры заключается в возможности обновления (замены) модуля, без необходимости изменения остальной системы, упрощении тестирование программы и обнаружение ошибок, упрощении задачи проектирования программы и распределения процесса разработки между группами разработчиков. При разбиении программы на модули для каждого модуля указывается реализуемая им функциональность, а также связи с другими модулями.

Важнейшим принципом разработки модуля является принцип информационной закрытости. Этот принцип утверждает, что содержание модулей должно быть скрыто друг от друга. Модуль должен определяться и проектироваться так, чтобы его содержимое (функции и данные) было недоступно тем модулям, которые не нуждаются в такой информации.

Модуль в языке Си состоит из интерфейса (заголовочого файла .h) и реализации (файла .c).

Программный код разбивается на несколько файлов, каждый из которых компилируется отдельно от остальных. Такая модульность программного кода позволяет значительно уменьшить время перекомпиляции при изменениях, вносимых лишь в небольшое количество исходных файлов, и упрощает групповую разработку. Также это возможность замены отдельных компонентов конечного программного продукта, без необходимости пересборки всего проекта.

Файл как единица трансляции не должен содержать более одного определения любой переменной, функции, структуры, перечисления. А вот прототипов функций (объявлений) может быть несколько. Объявление лишь добавляет некоторое имя в данную область видимости и обычно используется для согласования типов.

Чтобы избежать повторного определения переменных в другом модуле, в С++ включено специальное ключевое слово extern.

1. Подпрограммы. Преимущества и недостатки использования. Виды.

С ростом размера программы появляется необходимость разбивать решаемую задачу на подзадачи и оформлять их в виде отдельных частей (подпрограмм). В этом случае основная программа служит только для вызова этих подпрограмм.

Подпрограмма — это часть всей программы, оформленная особым образом. Как правило, логически завершённая часть программы. Активное использование подпрограмм при разработке программного обеспечения составляет основу модульного программирования.

Главное отличие подпрограммы от основной (главной) программы заключается в том, что управление может быть передано только главной программе. *Подпрограмма может быть откомпилирована, но не может быть запущена на исполнение*.

**Преимущества от использования подпрограмм**:

1. Возможность создания достаточно больших программ.
2. Достаточно просто повторно использовать ранее написанный код.
3. Разработку отдельных частей программы, т.е. подпрограмм, можно поручить разным людям.
4. Сокращается срок разработки программы.
5. Повышается надёжность программы.
6. В ряде случаев уменьшается размер программы.

**Недостатки от применения подпрограмм**:

1. Использование подпрограмм всегда уменьшает скорость работы программы.
2. Размер программы не всегда, но часто возрастает.
3. Требуется более высокая квалификация от программиста.

***Виды подпрограмм***

Существует две категории подпрограмм: процедуры и функции.

Процедура — это подпрограмма, которая не возвращает через своё имя результата работы, поэтому она вызывается как отдельный оператор. Процедура «общается с внешним миром» через список параметров. Часть из них будут входными, часть — выходными.

Функция — это подпрограмма, которая через имя возвращает результат своей работы. Функция вызывается в выражении, а не как отдельный оператор. Через список параметров она может получить входные данные и вернуть результаты работы.

Традиционно подпрограмму оформляют в виде функции, если результатом работы является одиночный объект.

Если результатом работы подпрограммы является несколько объектов, или выходным данным является массив, то необходимо использовать процедуру, так как через имя подпрограммы можно вернуть только один простой объект, а через список параметров — любое количество и одиночных и составных объектов.

1. Определение и вызов функций. Фактические и формальные параметры. Передача параметров в функции по значению, по ссылке, по указателю.

**Функция** – это последовательность операторов, оформленная таким образом, что ее можно вызвать по имени из любого места программы.

Объявление функции (прототип) задает ее имя, тип возвращаемого значения, список передаваемых параметров, необязательный спецификатор памяти (extern, static) и завершается символом «точка с запятой».

Определение функции содержит заголовок функции и ее тело, представляющее собой программный код, выполняемый при вызове функции (последовательность операторов и описаний в фигурных скобках).

В общем виде функция выглядит следующим образом:

[класс памяти] *Возвращаемый тип* имя функции (список параметров) { тело функции }

Возвращаемый *тип* определяет тип данного, возвращаемого функцией. Список параметров — это список, элементы которого отделяются друг от друга запятыми. Каждый такой элемент состоит из имени переменной и ее типа данных. Класс памяти (необязательно) – extern (глобальная видимость во всех модулях программы) и static (видимость только в пределах модуля, в котором определена функция).

Существенное различие между определением и объявлением функции: определений может быть несколько. Объявление функции должно находиться в тексте раньше ее вызова для того, чтобы компилятор мог осуществить проверку правильности вызова.

**Формальные параметры** существуют в прототипе и теле определения функции. Они задаются некоторыми уникальными именами и внутри функции доступны как локальные переменные. **Фактические параметры** существуют в основной программе. Они указываются при вызове функции на месте формальных.

В момент вызова функции значения фактических параметров присваиваются формальным. Соответственно, имена формальных и фактических параметров могут совпадать, это не вызовет конфликта.

**Передача параметров в функции по значению, по ссылке, по указателю**

Передача параметров по значению

При его применении в формальный параметр подпрограммы копируется значение аргумента. В таком случае изменения параметра на аргумент не влияют. Код, находящийся внутри функции, не может изменять значений аргументов, которые использовались при вызове функции.

В момент обращения к функции в памяти создаются временные переменные с именами, указанными в списке параметров, в которые копируются значения фактических (передаваемых в функцию) параметров. После завершения работы функции временные переменные удаляются из памяти.

Передача параметров по ссылке (&)

При передаче параметров по ссылке передается адрес соответствующей переменной, а не ее значение. Для получения адреса используется операция разадресации («&»).

При его применении в параметр копируется адрес аргумента. Это значит, что изменения значения параметра приводят к точно таким же изменениям значения аргумента.

Передача параметров по указателю (второе имя)

Так же как и при передаче параметров по ссылке, в данном способе используется не значение соответствующей переменной, а ее адрес. Отличие от предыдущего способа состоит в том, что используется операция косвенной адресации (\*).

void Swap\_Ptr (double \*a, double \*b) { ... }

Swap\_Ptr (&x, &y); // Вызов функции и явное разыменование переменных в функции (\*a и \*b).

1. Функции. Возвращаемые значения. Глобальные и локальные данные.

Функция может принимать параметры и возвращать значение.

Разделение программы на функции позволяет также избежать избыточности кода, поскольку функцию записывают один раз, а вызывать ее на выполнение можно из разных точек программы многократно. Функция начинает выполняться в момент вызова.

**Возвращаемые значения.**

Все функции, кроме функций типа void, возвращают значения. Функция может возвращать любые типы данных, кроме массивов. Данное значение определяется в операторе return. Если функция не определена как void и если не указано возвращаемое значение, то возвращается мусор. Если функция не объявлена как void, она может использоваться в качестве операнда в любом корректном выражении, иначе - не может.

В противоположность **локальным** переменным **глобальные** переменные видны всей программе и могут использоваться любым участком кода. Они хранят свои значения на протяжении всей работы программы. Глобальные переменные создаются путем объявления вне функции. К ним можно получить доступ в любом выражении, независимо от того, в какой функции находится данное выражение.

Глобальные переменные хранятся в фиксированной области памяти, устанавливаемой компилятором. Глобальные переменные чрезвычайно полезны, когда одни и те же данные используются в нескольких функциях программы. Следует избегать ненужного использования глобальных переменных по трем причинам:

* Они используют память в течение всего времени работы программы, а не тогда, когда они необходимы.
* Использование глобальных переменных вместо локальных приводит к тому, что функции становятся более частными, поскольку они зависят от переменных, определяемых снаружи.
* Использование большого числа глобальных переменных может вызвать ошибки в программе из-за неизвестных и нежелательных эффектов.

1. Функции. Параметры со значением по умолчанию. Способы передачи параметров

**Параметры со значением по умолчанию**

При объявлении функции для некоторых аргументов можно задавать значение по умолчанию, которое передается в функцию в случае, если при вызове соответствующий параметр не задан. Так как компилятор присваивает имеющиеся значения последовательно слева направо, то аргументы, имеющие заданное по умолчанию значение, должны располагаться правее аргументов, не имеющих такого значения.

Параметры по умолчанию должны быть указаны при первом упоминании функции (в прототипе).

**Передача параметров в функции по значению, по ссылке, по указателю**

При работе важно соблюдать следующее правило: при объявлении и вызове функции параметры должны соответствовать по количеству, порядку следования и типам. Функция может не иметь параметров, в этом случае после имени функции обязательно ставятся круглые скобки. Существует три основных способа передачи параметров: передача по значению, по ссылке или по указателю. (см. вопр. 17) :)

1. Функции. С переменным числом параметров.

При вызове функции с переменным числом параметров в вызове этой функции задается любое требуемое число аргументов. В объявлении и определении такой функции переменное число аргументов задается многоточием в конце списка формальных параметров или списка типов аргументов.

Все аргументы, заданные в вызове функции, размещаются в стеке. Количество формальных параметров, объявленных для функции, определяется числом аргументов, которые берутся из стека и присваиваются формальным параметрам. Программист отвечает за правильность выбора дополнительных аргументов из стека и определение числа аргументов, находящихся в стеке.

Формат объявления функции с переменным числом параметров:

тип\_возвращаемого\_значения имя\_функции (список\_параметров, …)

Список параметров содержит хотя бы один обязательный параметр. Многоточие указывает на возможность добавления любого числа параметров.

Для работы с параметрами определен тип списка va\_list и три макроса:

void va\_start(va\_list указатель, имя\_послед. \_обязат.\_аргумента)

начинает работу со списком. Устанавливает указатель на первый необязательный аргумент.

void va\_arg(va\_list указатель, тип\_аргумента)

возвращает значение очередного аргумента из списка. Каждый запуск макроса переводит указатель на следующий аргумент. Достижение последнего аргумента списка не контролируется.

void va\_end(va\_list указатель)

завершает работу со списком и освобождает память.

1. Функции. Указатели на функции.

Указатели на функции— очень мощное средство языка С. Функция располагается в памяти по определенному адресу, который можно присвоить указателю в качестве его значения. Адресом функции является ее точка входа. Именно этот адрес используется при вызове функции. Так как указатель хранит адрес функции, то она может быть вызвана с помощью этого указателя. Он позволяет также передавать ее другим функциям в качестве аргумента. Имя функции является константным указателем на начало функции в оперативной памяти.

Например, имеется функция

double y(double x, int n)

{

Тело\_функции

}

Указатель на такую функцию имеет вид:

double (\*fun)(double, int);

Скобки вокруг fun необходимы для правильной интерпретации объявления компилятором.

Если присвоить указателю fun адрес функции y: fun = y;

то функцию можно вызывать

x = fun(t, m);

Массив указателей на функции можно использовать в разных целях. Один из вариантов такого использования – можно похожие функции собрать в массив и потом уже вызывать из массива.

void a() {cout<<"A";} //функция а

void b() {cout<<"B";} //функция b

void c() {cout<<"C";} //функция с

int main()

{

void (\*P[3])()={a,b,c}; (\*P[2])(); cin.get();

 return 0;}

1. Параметры функции main()

Обычно информация, передаваемая при запуске программы, передается функции main() с помощью аргументов командной строки. Аргумент командной строки — это информация, которая вводится в командной строке операционной системы вслед за именем программы.

Чтобы принять аргументы командной строки, используются два специальных встроенных аргумента: argc и argv. Параметр **argc** содержит количество аргументов в командной строке и является целым числом, причем он всегда не меньше 1, потому что первым аргументом считается имя программы. А параметр **argv** является указателем на массив указателей на строки. В этом массиве каждый элемент указывает на какой-либо аргумент командной строки.

Первый индекс argv обеспечивает доступ к строке, а второй индекс — доступ к ее отдельным символам.

Обычно argc и argv используют для того, чтобы передать программе начальные команды, которые понадобятся ей при запуске. Например, аргументы командной строки часто указывают такие данные, как имя файла, параметр или альтернативное поведение.

Имена argc и argv являются традиционными, но не обязательными. Эти два параметра в функции main() вы можете назвать как угодно.

1. Функции. Рекурсивные функции.

В языке С функция может вызывать сама себя. В этом случае такая функция называется рекурсивной. **Рекурсия** — это процесс определения чего-либо на основе самого себя.

**Минусы.** Во многих случаях скорость медленнее. Много рекурсивных вызовов в функции может привести к переполнению стека.

**Плюсы.** Использование их для более простого создания версии некоторых алгоритмов по сравнению с итеративными эквивалентами.

**Рекомендации.** При написании рекурсивных функций следует иметь оператор if, чтобы заставить функцию вернуться без рекурсивного вызова. Если это не сделать, то, однажды вызвав функцию, выйти из нее будет невозможно

**Виды.** Рекурсию можно назвать простой, если в функции присутствует лишь один рекурсивный вызов. Такую рекурсию можно назвать еще рекурсией первого порядка. Но рекурсивный вызов может появляться в функции более, чем один раз. В таких случаях можно выделить следующие виды рекурсии:

* параллельная рекурсия. (defun function\_1 … (function\_2 … (function\_1 …) … (function\_1 …) … ) … )
* взаимная рекурсия. (defun function\_1 … (function\_2 … ) … ) (defun function\_2 … (function\_1 … ) … )
* рекурсия более высокого порядка. (defun function\_1 … (function\_1 … (function\_1 …) … ) … )

**Память.** Когда функция вызывает сама себя, новый набор локальных переменных и параметров размещается в памяти в стеке, а код функции выполняется с самого своего начала, причем используются именно эти новые переменные. При рекурсивном вызове функции новая копия ее кода не создается. Новыми являются только значения, которые использует данная функция. При каждом возвращении из рекурсивного вызова старые локальные переменные и параметры извлекаются из стека, и сразу за рекурсивным вызовом возобновляется работа функции.

1. Динамические структуры данных. Рекомендации для использования. Основные особенности.

Под динамической структурой данных понимается любая структура данных, занимаемый объем памяти которой не является фиксированным. Размер подобной структуры ограничен только объемом оперативной памяти компьютера.

Динамические структуры данных бывают двух видов:

• несвязанные динамические данные;

• связанные динамические данные.

Несвязанные динамические данные размещаются в памяти программистом, а не автоматически. К связанным динамическим данным относятся списки, очереди и стеки; это комбинированные данные, элементы которых связаны между собой с помощью адресных ссылок.

Особенности динамической структура:

• она не имеет имени;

• ей выделяется память в процессе выполнения программы;

• количество элементов структуры может не фиксироваться;

• размерность структуры может меняться в процессе выполнения программы;

• в процессе выполнения программы может меняться характер взаимосвязи между элементами структуры.

Рекомендуется использовать динамические структуры в следующих случаях:

• Используются переменные, имеющие довольно большой размер (массивы большой размерности), необходимые в одних частях программы и совершенно не нужные в других.

• В процессе работы программы нужен массив, список или иная структура, размер которой изменяется в широких пределах и трудно предсказуем.

• Когда размер данных, обрабатываемых в программе, превышает объем сегмента данных.

Создание и использование динамических структур данных требует динамического распределения памяти – возможности получать в процессе исполнения дополнительную память для хранения новых узлов и освобождать блоки памяти, ставшие ненужными.

1. Стек. Создание стека. Общие принципы работы

**Стек** – это структура данных, в которой новый элемент всегда записывается в ее начало (вершину) и очередной читаемый элемент также всегда выбирается из ее начала. В стеках используется метод доступа к элементам *LIFO* ( *Last Input – First Output*, "последним пришел – первым вышел").

На стек выделяется ограниченная область памяти. При каждом вызове процедуры в стек добавляются новые элементы (параметры, локальные переменные, адрес возврата). Поэтому если вложенных вызовов будет много, стек переполнится.

Стек – это список, у которого доступен один элемент (одна позиция). Этот элемент называется *вершиной стека*. Взять элемент можно только из вершины стека, добавить элемент можно только в вершину стека.

Описание стека выглядит следующим образом:

struct имя\_типа {

информационное поле;

адресное поле;

};

struct list {

type pole1;

list \*pole2;

} stack;

где информационное поле – это поле любого ранее объявленного или стандартного типа;

адресное поле – это указатель на объект того же типа, что и определяемая структура, в него записывается адрес следующего элемента стека.

Стек как динамическую структуру данных легко организовать на основе линейного списка. Для такого списка достаточно хранить указатель вершины стека, который указывает на первый элемент списка. Если стек пуст, то списка не существует, и указатель принимает значение NULL.

1. Стек. Удаление элемента. Добавление элемента. Общие принципы работы

**Основными операциями над стеками являются:**

* добавление элемента;
* удаление элемента;
* чтение верхнего элемента.

Методы:

* **push()** – добавить элемент;
* **pop()** – удалить элемент;
* **top()** – получить верхний элемент;
* **size()** – размер стека;
* **empty()** – проверить стек на наличие элементов.

**Создание/добавление.**

При добавлении элемента у нас возникнет две ситуации:

* Стек пуст, и нужно создать его
* Стек уже есть и нужно лишь добавить в него новый элемент

void s\_push(comp \*\*top, int D) { *// указатель на вершину стека и переменную записываемая в ячейку*

comp \*q; *// новый указатель q типа структуры comp. наш новый элемент*

q = new comp(); *//выделяем память для нового элемента*

q->Data = D; *//Записываем необходимое число в Data элемента*

if (top == NULL) { *//стек пустой*

\*top = q; *//вершиной стека будет новый элемент*

}

else

{

q->next = \*top; *// кладем на вершину стопки.*

\*top = q; *//вершиной теперь является новый элемент*

}

}

**Удаление.**

Здесь могут быть такие варианты:

* Ячейка, которую нам нужно удалить является вершиной стека
* Ячейка, которую нам нужно удалить находится в конце, либо между двумя ячейками

void s\_delete\_key(comp \*\*top, int N) {*// принимает вершину top и число которое нужно удалить*

comp \*q = \*top; *//создаем указатель и ставим его на вершину стека*

comp \*prev = NULL;*//создаем указатель на предыдуший элемент*

while (q != NULL)

if (q->Data == N) {

if (q == \*top) {*//если элемент, который нам нужно удалить - вершина*

\*top = q->next;*//передвигаем вершину на следующий элемент*

free(q);*//очищаем ячейку*

q->Data = NULL; *// обнуляем переменные в удаленной ячейке,*

q->next = NULL;

}

else*//если элемент последний или находится между двумя другими элементами*

{

prev->next = q->next;*// связь от предыдущего элемента к следующему*

free(q);*//очищаем ячейку*

q->Data = NULL;*//обнуляем переменные*

q->next = NULL;

}

}*// если Data элемента НЕ равна числу, которое нам нужно удалить*

prev = q; *//запоминаем текущую ячейку как предыдущую*

q = q->next;*//перемещаем указатель q на следующий элемент*

}

}

Так же стоит отметить, что если не провести данную связь, участок ячеек, который лежит после удаленной ячейки станет недоступным, так как потеряется та самая связь, которая соединяет одну ячейку с другой и данный участок просто затеряется в памяти

1. Очередь. Создание очереди.

**Очередь** – это структура данных, представляющая собой последовательность элементов, образованная в порядке их поступления. В очереди используется принцип доступа к элементам *FIFO* ( *First Input – First Output*, "первый пришёл – первый вышел"). В очереди доступны два элемента (две позиции): *начало очереди* и *конец очереди*. Поместить элемент можно только в конец очереди, а взять элемент только из ее начала.

Существует несколько способов реализации очереди:

* с помощью одномерного массива;
* с помощью связанного списка;
* с помощью класса объектно-ориентированного программирования.

Описание очереди выглядит следующим образом:

struct имя\_типа {

информационное поле;

адресное поле1;

адресное поле2;

};

struct list2 {

type pole1;

list1 \*beg, \*next ;

}

где информационное поле – это поле любого, ранее объявленного или стандартного, типа; адресное поле1, адресное поле2 – это указатели на объекты того же типа, что и определяемая структура, в них записываются адреса первого и следующего элементов очереди.

Очередь как динамическую структуру данных легко организовать на основе линейного списка. Поскольку работа идет с обоими концами очереди, то предпочтительно будет использовать линейный двунаправленный список. Хотя для работы с таким списком достаточно иметь один указатель на любой элемент списка, здесь целесообразно хранить два указателя – один на начало списка (откуда извлекаем элементы) и один на конец списка (куда добавляем элементы). Если очередь пуста, то списка не существует, и указатели принимают значение NULL.

1. Очередь. Удаление элемента. Добавление элемента. Общие принципы работы

Стандартный набор операций:

* добавление элемента;
* удаление элемента;
* чтение первого элемента.

Методы:

* **push()** – добавить элемент;
* **pop()** – удалить первый элемент;
* **front()** – обратиться к первому элементу;
* **back()** – обратиться к последнему элементу;
* **empty()** – проверить пуста ли очередь.

Реализация на основе массива.

struct queue {

int\*v;

int head;

int tail;

Int size;

Int maxsize;

};

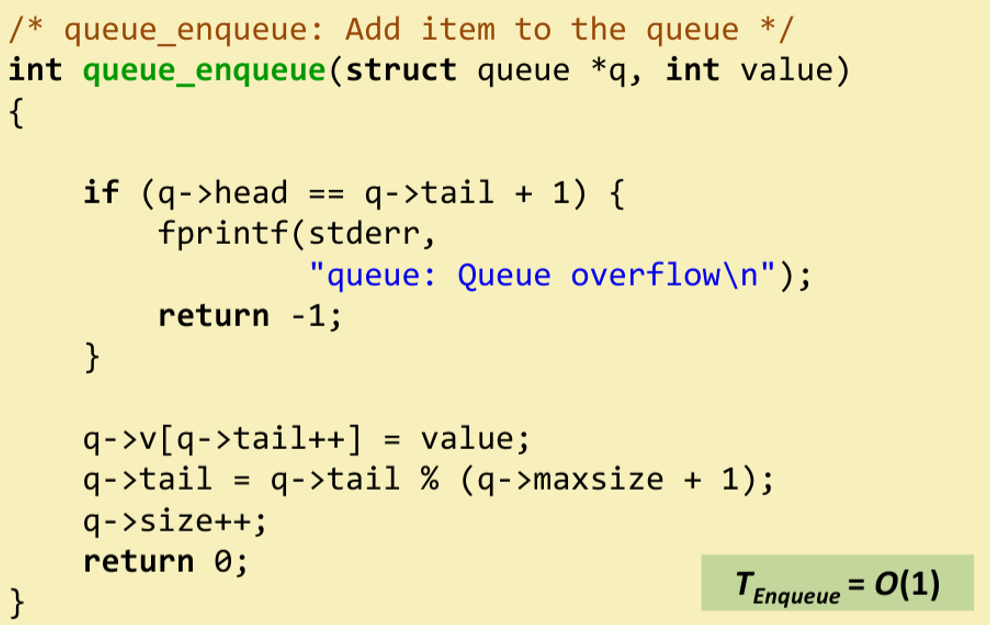
void queue\_free (struct queue \*q)

{

free(q->v);

free(q);

};



1. Линейные списки. Создание элемента списка.

**Списком** называется упорядоченное множество, состоящее из переменного числа элементов, к которым применимы операции включения, исключения. Список, отражающий отношения соседства между элементами, называется *линейным*.

*Длина списка* равна числу элементов, содержащихся в списке, список нулевой длины называется *пустым списком*. Списки представляют собой способ организации структуры данных, при которой элементы некоторого типа образуют цепочку. Для связывания элементов в списке используют систему указателей. В минимальном случае, любой элемент линейного списка имеет один указатель, который указывает на следующий элемент в списке или является пустым указателем, что интерпретируется как конец списка.

Наиболее простой динамической структурой является однонаправленный список, элементами которого служат объекты структурного типа. Представляет собой последовательность элементов, в каждом из которых хранится значение и указатель на следующий элемент списка. В последнем элементе указатель на следующий элемент равен NULL.

struct имя\_типа {

информационное поле;

адресное поле;

};

struct Node {

int key;//информационное поле

Node\*next;//адресное поле

};

где информационное поле – это поле любого, ранее объявленного или стандартного, типа; адресное поле – это указатель на объект того же типа, что и определяемая структура, в него записывается адрес следующего элемента списка. Информационных полей может быть несколько.

Каждый элемент списка содержит ключ, который идентифицирует этот элемент. Ключ обычно бывает либо целым числом, либо строкой.

Создание.

Для того, чтобы создать список, нужно создать сначала первый элемент списка, а затем при помощи функции добавить к нему остальные элементы.

1. Линейные списки. Добавление узла в начало списка.

template< typename T >

void List< T >::append( const T &t ) {

// Создаем новый узел для значения

// Не забудем проверить, что память удалось выделить

if( Node\* node = new Node( t ) ) {

// Новый узел привязывается к старому головному элементу

node->m\_next = m\_head;

// Новый узел сам становится головным элементом

m\_head = node;

}

}

1. Линейные списки. Поиск элемента.

Операция поиска элемента в списке заключается в последовательном просмотре всех элементов списка до тех пор, пока текущий элемент не будет содержать заданное значение или пока не будет достигнут конец списка. В последнем случае фиксируется отсутствие искомого элемента в списке (функция принимает значение false). Входными параметрами являются значение, которое должен содержать искомый элемент и указатель на список. В качестве выходного параметра передается указатель, который устанавливается на найденный элемент или остается без изменений, если элемента в списке нет.

Алгоритм поиска можно реализовать рекурсивным и нерекурсивным способом. Рекурсивные решения обычно более понятны, но менее оптимальны.

### **Решение 1. Размер связного списка известен**

Если размер связного списка известен, k-й элемент с конца легко вычислить (длина – k). Нужно пройтись по списку и найти этот элемент.

### **Решение 2. Рекурсивное решение**

Такой алгоритм рекурсивно проходит связный список. По достижении последнего элемента алгоритм начинает обратный отсчет, и счетчик сбрасывается в 0. Каждый шаг инкрементирует счетчик на 1. Когда счетчик достигнет k, искомый элемент будет найден.

Реализация этого алгоритма коротка и проста – достаточно передать назад целое значение через стек. К сожалению, оператор return не может вернуть значение узла.

**Подход А: не возвращайте элемент**

Можно не возвращать элемент, достаточно вывести его сразу, как только он будет найден. А в операторе return вернуть значение счетчика.

**Подход Б: используйте C++**

Второй способ – использование передачу значения по ссылке. Такой подход позволяет не только вернуть значение узла, но и обновить счетчик путем передачи указателя на него.

1. Линейные списки. Удаление узла.

В качестве аргументов функции удаления элемента ОЛС передаются указатель на удаляемый узел, а также указатель на корень списка.  
Функция возвращает указатель на узел, следующий за удаляемым.  
Удаление узла ОЛС включает в себя следующие этапы:

* установка указателя предыдущего узла на узел, следующий за удаляемым;
* освобождение памяти удаляемого узла.

struct list \* deletelem(list \*lst, list \*root)  
{  
  struct list \*temp;  
  temp = root;  
  while (temp->ptr != lst) // просматриваем список начиная с корня  
  { // пока не найдем узел, предшествующий lst  
    temp = temp->ptr;  
  }  
  temp->ptr = lst->ptr; // переставляем указатель  
  free(lst); // освобождаем память удаляемого узла  
  return(temp);  
}

1. Двусвязный список. Создание. Удаление узла.

Для ускорения многих операций целесообразно применять переходы между элементами списка в обоих направлениях. Это реализуется с помощью двунаправленных списков, которые являются сложной динамической структурой.

**Двунаправленный (двусвязный) список** – это структура данных, состоящая из последовательности элементов, каждый из которых содержит информационную часть и два указателя на соседние элементы. При этом два соседних элемента должны содержать взаимные ссылки друг на друга.

В таком списке каждый элемент (кроме первого и последнего) связан с предыдущим и следующим за ним элементами. Наличие ссылок на следующее звено и на предыдущее позволяет двигаться по списку от каждого звена в любом направлении: от звена к концу списка или от звена к началу списка, поэтому такой список называют двунаправленным.

Описание простейшего элемента такого списка выглядит следующим образом:

struct имя\_типа {

информационное поле;

адресное поле 1;

адресное поле 2;

};

где информационное поле – это поле любого, ранее объявленного или стандартного, типа; адресное поле 1 – это указатель на объект того же типа, что и определяемая структура, в него записывается адрес *следующего элемента списка* ; адресное поле 2 – это указатель на объект того же типа, что и определяемая структура, в него записывается адрес *предыдущего элемента списка*.

В отличие от однонаправленного списка здесь нет необходимости обеспечивать позиционирование какого-либо указателя именно на первый элемент списка, так как можно получить доступ к любому элементу списка из любого другого элемента. Однако по правилам хорошего тона программирования указатель желательно ставить на заголовок списка.

**Создание двунаправленного списка**

Для того, чтобы создать список, нужно создать сначала первый элемент списка, а затем при помощи функции добавить к нему остальные элементы. Добавление может выполняться как в начало, так и в конец списка. Реализуем рекурсивную функцию.

**Удаление элемента из двунаправленного списка**

Операция удаления элемента из двунаправленного списка осуществляется во многом аналогично удалению из однонаправленного списка. В качестве аргументов функции удаления узла ДЛС передается указатель на удаляемый узел. Поскольку узел списка имеет поле указателя на предыдущий узел, нет необходимости передавать указатель на корень списка. Функция возвращает указатель на узел, следующий за удаляемым.

1. Двусвязный список. Создание. Добавление узла.

(см.вопр.33)

Функция добавления узла в список принимает два аргумента:

* Указатель на узел, после которого происходит добавление
* Данные для добавляемого узла.

Добавление узла в ДЛС включает в себя следующие этапы:

* создание узла добавляемого элемента и заполнение его поля данных;
* переустановка указателя «следующий» узла, предшествующего добавляемому, на добавляемый узел;
* переустановка указателя «предыдущий» узла, следующего за добавляемым, на добавляемый узел;
* установка указателя «следующий» добавляемого узла на следующий узел (тот, на который указывал предшествующий узел);
* установка указателя «предыдущий» добавляемого узла на узел, предшествующий добавляемому (узел, переданный в функцию).

\*Возвращаемым значением функции является адрес добавленного узла.

1. Деревья. Общие понятия.

**Дерево** – это совокупность узлов (вершин) и соединяющих их направленных ребер (дуг), причем в каждый узел (за исключением одного - корня) ведет ровно одна дуга.

**Корень** – это начальный узел дерева, в который не ведет ни одной дуги.

**Предком** для узла x называется узел дерева, из которого существует путь в узел x.

**Потомком** узла x называется узел дерева, в который существует путь (по стрелкам) из узла x.

**Родителем** для узла x называется узел дерева, из которого существует непосредственная дуга в узел x.

**Сыном** узла x называется узел дерева, в который существует непосредственная дуга из узла x.

**Уровнем** узла x называется длина пути (количество дуг) от корня к данному узлу. Считается, что корень находится на уровне 0.

**Листом** дерева называется узел, не имеющий потомков.

**Внутренней** **вершиной** называется узел, имеющий потомков.

**Высотой** дерева называется максимальный уровень листа дерева.

**Упорядоченным** **деревом** называется дерево, все вершины которого упорядочены.

дерево – это корень и несколько связанных с ним деревьев (поддеревьев). Размер памяти, необходимый для хранения дерева, заранее неизвестен, потому что неизвестно, сколько узлов будет в него входить. На практике используются главным образом деревья особого вида, называемые двоичными (бинарными).

**Двоичным** **деревом** называется дерево, каждый узел которого имеет не более двух сыновей. Двоичные деревья упорядочены, то есть различают левое и правое поддеревья. Пример использования - когда на каждом этапе некоторого процесса надо принять одно решение из двух возможных.

**Строго** **двоичным** деревом называется дерево, у которого каждая внутренняя вершина имеет непустые левое и правое поддеревья.

**Полным** **двоичным** деревом называется дерево, у которого все листья находятся на одно уровне и каждая внутренняя вершина имеет непустые левое и правое поддеревья.

1. Бинарные деревья

(см.вопр.35)

**Бинарное (двоичное) дерево** – это динамическая структура данных, представляющее собой дерево, в котором каждая вершина имеет не более двух потомков. Таким образом, бинарное дерево состоит из элементов, каждый из которых содержит информационное поле и *не более двух* ссылок на различные бинарные поддеревья. На каждый элемент дерева имеется ровно одна ссылка.

Каждая вершина бинарного дерева является структурой, состоящей из четырех видов полей. Содержимым этих полей будут соответственно:

* информационное поле (ключ вершины);
* служебное поле (их может быть несколько или ни одного);
* указатель на левое поддерево;
* указатель на правое поддерево.

1. Обход бинарного дерева.

**Обход дерева** – это упорядоченная последовательность вершин дерева, в которой каждая вершина встречается только один раз. При обходе все вершины дерева должны посещаться в определенном порядке.

В зависимости от траекторий выделяют два типа обхода:  
— горизонтальный (в ширину);   
— вертикальный (в глубину).

**Горизонтальный обход** подразумевает обход дерева по уровням (level-ordered) – вначале обрабатываются все узлы текущего уровня, после чего осуществляется переход на нижний уровень.  
При **вертикальном обходе** порядок обработки текущего узла и узлов его правого и левого поддеревьев варьирует и по этому признаку выделяют три варианта вертикального обхода:

— прямой (префиксный, pre-ordered): вершина – левое поддерево – правое поддерево;   
— обратный (инфиксный, in-ordered): левое поддерево – правое поддерево – вершина  
— симметричный (постфиксный, post-ordered): левое поддерево – вершина – правое поддерево;

1. Бинарные деревья. Поиск в дереве.

**Поиск в ширину** (BFS) идет из начальной вершины, посещает сначала все вершины находящиеся на расстоянии одного ребра от начальной, потом посещает все вершины на расстоянии два ребра от начальной и так далее. Алгоритм поиска в ширину является по своей природе не рекурсивным (итеративным). Для его реализации применяется структура данных очередь (FIFO).  
**Поиск в глубину** (DFS) идет из начальной вершины, посещая еще не посещенные вершины без оглядки на удаленность от начальной вершины. Алгоритм поиска в глубину по своей природе является рекурсивным. Для эмуляции рекурсии в итеративном варианте алгоритма применяется структура данных стек.

1. Сбалансированные деревья

АВЛ-дерево — это двоичное дерево поиска, ключи которого удовлетворяют свойству: ключ любого узла дерева не меньше любого ключа в левом поддереве данного узла и не больше любого ключа в правом поддереве этого узла. Это значит, что для поиска нужного ключа можно использовать стандартный алгоритм.  
Особенностью АВЛ-дерева является то, что оно является сбалансированным в следующем смысле: для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу. А так как основные операции над двоичными деревьями поиска (поиск, вставка и удаление узлов) линейно зависят от его высоты, то получаем *гарантированную* логарифмическую зависимость времени работы этих алгоритмов от числа ключей, хранимых в дереве. Вероятность получения сильно несбалансированного дерева при больших к-вах узлов хотя и является пренебрежимо малой, но остается *не равной нулю*.

1. Файлы. Типы файлов.

Файл − поименованная совокупность данных, расположенных на внешнем носителе.

Различают два вида файлов: текстовые и двоичные.

**Текстовые файлы** хранят информацию в виде последовательности символов. Вывод осуществляется аналогично выводу на экран. Текстовые файлы могут быть просмотрены и отредактированы с клавиатуры любым текстовым редактором и имеют очень простую структуру: последовательность ASCII-символов. Текстовые файлы разбиваются на несколько строк с помощью специального символа "конец строки". Текстовый файл заканчивается специальным символом "конец файла".

**Бинарные (или двоичные) файлы** предназначены для хранения последовательности байтов. Структура такого файла определяется программно. Каждая программа для своих бинарных файлов определяет собственную структуру.  При записи в двоичный файл символы и числа записываются в виде последовательности байт (в своем внутреннем двоичном представлении в памяти компьютера).

При работе с файлами используются следующие макросы:

- NULL − определяет пустой указатель;

- EOF – значение, возвращаемое при попытке чтения после конца файла;

- FOPEN\_MAX – возвращает максимальное число одновременно открытых файлов.

По умолчанию файл открывается в текстовом режиме.

1. Доступ к файлам.

Различают два вида файлов: текстовые и двоичные. В отличие от текстовых файлов доступ к элементам бинарных файлов выполняется в произвольном порядке, а не последовательно.

Большинство ф-ий для работы с файлами находится в библиотеке stdio.h.

Каждому файлу присваивается внутреннее логическое имя, используемое в дальнейшем при обращении.

**FILE\* указатель\_на\_файл;**

В начале работы для получения доступа к данным файл необходимо открыть.

**FILE\* fopen(char\* ID\_файла, char\* режим);**

При открытии файла указатель текущей позиции помещается в начало файла. При ошибке открытия возвращается NULL. После выполнения любой операции над данными указатель сдвигается на одну позицию вперед. В конце работы файл закрывается, т. е. доступ к данным, размещенным в файле, будет запрещен.

**Режимы открытия:**

* **"w"**: файл открывается для записи. Если файл ранее существовал, информация удаляется.
* **"r"**: файл открывается для чтения
* **"a"**: файл открывается для добавления в конец новых данных.
* **"w+"**: как r+.
* **"r+"**: файл открывается для редактирования данных (запись + чтение).
* **"a+"**: как а, только запись можно выполнять в любое место файла, доступно чтение файла.
* **"b"**: файл открывается в бинарном режиме.
* **"t"**: файл открывается в текстовом режиме.

Для закрытия файла используется функция **fclose(указатель файла);**. Для закрытия нескольких файлов используется функция **fcloseall(void)**.

Для изменения режима доступа:

* Закрытие
* Открытие в другом режиме

Используется функция **FILE\* freopen(char\* ID, char\* режим, FILE\* указатель);**

Создается временный файл с правами доступа “w+b”, автоматически удаляется после завершения работы:

**FILE\* tmp file(void);**

1. Файлы. Текстовые файлы. Открытие \ закрытие. (см.вопр. 40, 41)
2. Текстовые файлы. Ввод \ вывод в текстовые файлы. (см.вопр. 40) (см.вопр.41 – открытие)

Действия чтения-записи:

* Операции посимвольного ввода-вывода

int fgetc (FILE\* f) – считывает и возвращает символ из файла

int fputc (int ch, FILE\* f) – записывает в файл f код ch символа

* Операции построчного ввода-вывода

int fgets (char\* S, int m, FILE\* f) – чтение из файла f в строку S m-байт

int fputs (char\* S, FILE\* f) – запись в файл f строки S пока не встретиться ‘\0’

* Операции ввода-вывода по блокам

int fread (void\* p, int size, int n, FILE\* f) – считывает n блоков по size-байт из файла f в область памяти с указателем p.

int fwrite (void\* p, int size, int n, FILE\* f) – записывает n-блоков по size-байт каждый из области памяти с указателем p в файл f.

int fscanf (FILE\* f, char\* формат, список\_адресов\_объектов) – считывает из f информацию для объектов в соответствии с указанными форматами.

int fprintf (FILE\* f, char\* формат, список \_объектов) – записывают в f объекты, указанные в списке в соответствии с форматами.

1. Файлы. Бинарные файлы. Открытие \ закрытие файла. (см.вопр. 40, 41)
2. Бинарные файлы. Функции чтения \ записи. (см.вопр. 40, 41 - открытие)

Удобнее всего использовать функции **fread()** и **fwrite()**. (см.вопр. 43)

1. Бинарные файлы. Позиционирование в файле. (см.вопр.40)

Каждый открытый файл имеет, так называемый указатель на текущую позицию в файле. Все операции над файлами (чтение и запись) работают с данными с этой позиции. При каждом выполнении функции чтения или записи указатель смещается на количество прочитанных или записанных байт, т.е. устанавливается сразу за прочитанным или записанным блоком данных в файле. В этом случае осуществляется так называемый последовательный доступ к данным. Но иногда необходимо читать или писать данные в произвольном порядке.

**int fileno (FILE\* f)** – возвращает значение дескриптора файла f.

**long filelength (int fd)** – возвращает длину файла, имеющего номер (дескриптор) fd в байтах.

**int chsize (int fd, long pos)** – изменяет размер файла, имеющего номер fd, признак конца файла, устанавливается после байта с номером pos.

**int fseek (FILE\* f, long size, int kod)** – смещение указателя файла f на size-байт в направлении признака kod: 0 – от начала файла, 1 – от текущей позиции указателя, 2 – от конца файла.

**long ftell (FILE\* f)** – возвращает значение указателя на текущую позицию файла.

**int feof (FILE\* f)** – возвращает нулевое значение при правильной записи признака конца файла

**int fgetpos (FILE\* f, long\* pos)** – определяет значение текущей позиции pos файла f, возвращает 0 при успешном завершении.

1. Потоки. Виды потоков. Классы и объекты потоков ввода-вывода.

**Поток** – унифицированный интерфейс между программой и физ.устройствами, позволяющий выполнять перенос данных от источника к приемнику.

Чтение данных из потока называется **извлечением**, а вывод в поток – **помещением** или **включением**. Обмен с потоком для увеличения скорости передачи данных производится через буфер.

Потоки по направлению обмена:

* Входные
* Выходные
* Двунаправленные

Потоки по виду устройств, с которыми они работают:

* Стандартные (от клавиатуры на экран дисплея)
* Файловые (обмен информацией с файлами)
* Строковые (работа с массивами символов в ОП)

Ввод-вывод на **низком** **уровне** – каждый байт является самостоятельным элементом данных. Такой способ позволяет осуществить пересылку больших по объему потоков с высокой скоростью. Недостаток в трудоёмкости. При **высокоуровневом** потоковом вводе-выводе, байты группируются в значащие элементы данных. Неэффективен только для файлов очень большого объёма.

Библиотека iostream содержит много классов для обработки операций ввода-вывода. Например, класс istream поддерживает операции по вводу потоков, класс ostream поддерживает операции по выводу объектов, класс iostream поддерживает оба типа операций: и ввод, и вывод потоков.

Класс istream и класс ostream являются прямыми потомками класса ios. Класс iostream является производным классом множественного наследования классов istream и ostream.

Операции «поместить в поток» и «взять из потока» – это перегруженные операции сдвига влево (<<) и вправо (>>). Эти операции применяются к объектам стандартных потоков cin, cout, cerr и clog.

При обработке файлов C++ используются следующие классы:

–класс ifstream, который выполняет операции ввода из файлов;

–класс ofstream, который выполняет операции вывода в файлы;

–класс fstream, который выполняет операции ввода-вывода файлов. Класс ifstream наследует класс istream, класс ofstream наследует класс

ostream, а класс fstream – класс iostream.

1. Файловые потоки.

Описаны в заголовочном файле fstream.h.

Этапы работы с потоком:

* Создание файлового потока
* Открытие потока и связывание его с файлом
* Передача данных (ввод/вывод)
* Закрытие файла
* Уничтожение потока

Режимы открытия файла:

* ios::in: файл открывается для ввода.
* ios::out: файл открывается для вывода. При этом старые данные удаляются.
* ios::app: файл открывается для записи в конец. Старые данные не удаляются.
* ios::ate: после открытия файла перемещает указатель в конец файла
* ios::trunc: если файл существует, то его содержимое удаляется
* ios::binary: файл открывается в бинарном режиме

1. Средства работы с потоками ввода/вывода.

Методы ввода-вывода:

* get – посимвольный ввод.
* put – посимвольный вывод.
* getline – удаляет из входного потока символ перехода на новую строку.
* ignore – удаление 1 или нескольких текущих символов из входного потока.
* read/write – ввод-вывод нескольких байтов данных.
* gcount – получение к-ва байтов, введенное во время последней операции ввода.

seekg – изменение текущей позиции во входном файле.

seekp – в выходном файле:

* ios::beg – начало файла
* ios::cur – текущая позиция
* ios::end – конец файла

1. Форматирование ввода-вывода. Использование манипуляторов.

Для управления флагами:

* flags – состояние флагов потоков
* setf – установить флаги
* unsetf – сбросить флаги
* skwips – при вводе пробельные символы-разделители пропускаются
* left – выравнивание по левому краю
* right – выравнивание по правому краю
* internal – знак числа выводится по левому краю, число – по правому
* dec – целые значения ввод/вывод как 10-тичные числа
* oct – как 8-ричные
* hex – как 16-ричные

Для управления полем вывода:

* width – ширина поля вывода
* precision – точность
* fill – символ заполнения свободных позиций

Манипуляторы:

* endl
* flush – очистить все буферы
* dec, oct, hex
* setfill – установить символ заполнитель
* setw(int n)

1. Поиск. Линейный.

Для нахождения некоторого элемента (ключа) в заданном неупорядоченном массиве используется алгоритм линейного (последовательного) поиска. Он работает как с неотсортированными массивами, так и отсортированными.

Линейный поиск используется в случае, когда нет никакой дополнительной информации о местоположении разыскиваемых данных. Он представляет собой последовательный перебор массива до обнаружения требуемого ключа или до конца, если ключ не обнаружен.

Алгоритм: Начиная с первого, все элементы массива последовательно просматриваются и сравниваются с искомым. Если на каком-то шаге текущий элемент окажется равным искомому, тогда элемент считается найденным, и в качестве результата возвращается номер этого элемента, либо другая информация о нем. Иначе, следуют возвратить что-то, что может оповестить о его отсутствии в пройденной последовательности.

Код:

int LineSearch(int A[], int key)  
{  
for (i=0; i<N; i++)  
if (A[i]==key) return i;  
return -1;  
}

1. Поиск. Двоичный.

Двоичный(бинарный) поиск — алгоритм поиска элемента в отсортированном массиве. Бинарный поиск не используется для поиска максимального или минимального элементов, так как в отсортированном массиве эти элементы содержатся в начале и в конце массива соответственно.

Асимптотическая оценка алгоритма О(n) = log2n.

Метод основан на последовательном делении на 2 диапазона поиска. При этом на каждом шаге либо находится элемент, либо происходит переход в одну из половин диапазона. В процессе поиска выполняется не только сравнение на равенство, но и на больше - меньше. Последняя операция позволяет выбрать очередную половину диапазона таблицы.

Алгоритм: есть упорядоченный массив, берем число из середины массива, сравниваем с искомым. Если оно оказалось больше, значит искомое число в первой половине массива, если меньше — во второй. Продолжаем делить оставшуюся половину, когда находим нужное число возвращаем его индекс, если не находим возвращаем null.

int l = 0, r = n, find, mid, index;

cin >> find;

bool is\_find = false;

while (l<r) {

mid = (l + r) / 2;

if (find > A[mid]) {

l = mid + 1;

}

else if (find < A[mid]) {

r = mid - 1;

}

else if (find == A[mid]) {

index = mid;

is\_find = true;

break;

}

}

if (is\_find == true)

cout << index;

1. Пузырьковая сортировка.

Или **сортировка простыми обменами**. Принцип действий прост: обходим массив от начала до конца, попутно меняя местами неотсортированные соседние элементы. В результате первого прохода на последнее место «всплывёт» максимальный элемент. Теперь снова обходим неотсортированную часть массива (от первого элемента до предпоследнего) и меняем по пути неотсортированных соседей. Второй по величине элемент окажется на предпоследнем месте. Продолжая в том же духе, будем обходить всё уменьшающуюся неотсортированную часть массива, запихивая найденные максимумы в конец.

сложность: **O**(**n2**), количество сравнений \frac{(n^2-n)}{2}

1. Сортировка методом простого включения.

Алгоритм последовательного помещения элемента массива в отсортированную часть в соответствии с ключом сортировки.

Массив делят на 2 части:

* Отсортированную (готовую)
* Неотсортированную (исходную)

В начале работы: отсортированная часть – первый элемент массива, неотсортированная – все остальные элементы. На каждом шаге из неотсортированной части извлекается i-тый элемент и вставляется в отсортированную часть. Каждый шаг включает в себя 4 действия:

* Взять i-тый элемент массива, сохранить в дополнительной переменной
* Найти позицию j в отсортированной части, куда будет вставлен i-тый элемент
* Сдвиг элементов с i на одну позицию по j-ую
* Вставка элемента в j-ую позицию

Void InsertSort (int k, int x[max])

{ int I, j, temp;

for (i=o; i<k; i++)

{ temp = x[i];

for (j=i-1; (j>=0)&&(x[j]>temp); j--)

x[j+1]=x[j];

x[j+1]=temp;

}

}

1. Сортировка методом простого выбора.

Находим (выбираем) в массиве элемент с минимальным значением на интервале от 1-го элемента до n-го (последнего) элемента и меняем его местами с первым элементом. На втором шаге находим элемент с минимальным значением на интервале от 2-го до n-го элемента и меняем его местами со вторым элементом. И так далее для всех элементов до n-1-го.Таким образом, суть алгоритма сортировки выбором сводится к многократному поиску минимального (максимального) элементов в неотсортированной части массива. Худшее число сравнений и пересылок оцениваются О(n2). Алгоритм не использует дополнительной памяти.

for (int i = 0; i < N; i++) {

int min = i;

for (int j = i + 1; j < N; j++) {

if (arr[j] < arr[min]) {

min = j;

}

}

if (i != min) {

swap(arr[i], arr[min]);

}

}

1. Сортировка Хоара

Этот алгоритм, чаще называемый просто «быстрая сортировка» (англ. Quicksort), придуман английским ученым Чарльзом Хоаром в 1960 году и использует стратегию **«разделяй и властвуй»**.

Алгоритм:

* Выбор опорного элемента из массива
* Сравнение оставшихся элементов с опорным: меньше, больше, равны.
* Для обоих подмассивов, если в массиве больше 2-х элементов – рекурсивно запускаем ту же функцию

Поскольку в каждой итерации длина обрабатываемого отрезка массива уменьшается, терминальная ветвь рекурсии будет достигнута всегда и обработка гарантированно завершится.

Template <class T>

Void QuickSort (T\* a, long N)

{ long i=0, j=N-1;

T temp, p;

p = a[N>>1]

do

{ while (a[i]<p) i++

while (a[j]>p) j—

if (i<=j)

{temp=a[i]; a[i]=a[j]; a[j]=temp; i++, j--}

} while (i<=j)

if (j>0) QuickSort (a;j);

if (N>i) QuickSort (a+i; N-i);

}

1. Сортировка Шейкерная.

Сортировка перемешиванием ([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) Cocktailsort) — разновидность [пузырьковой сортировки](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%BF%D1%83%D0%B7%D1%8B%D1%80%D1%8C%D0%BA%D0%BE%D0%BC).

* если при движении по части массива перестановки не происходят, то эта часть массива уже отсортирована и, следовательно, ее можно исключить из рассмотрения.
* при движении от конца массива к началу минимальный элемент “всплывает” на первую позицию, а максимальный элемент сдвигается только на одну позицию вправо.

Эти две идеи приводят к следующим модификациям в методе пузырьковой сортировки. Границы рабочей части массива (где происходит движение) устанавливаются в месте последнего обмена на каждой итерации. Массив просматривается поочередно справа налево и слева направо.

int l = 0,

r = n - 1;

bool is\_sorted = false;

while (is\_sorted == false) {

is\_sorted = true;

for (int i = l; i < r; i++) {

if (A[i] > A[i + 1]) {

swap(A[i], A[i + 1]);

is\_sorted = false;

}

}

if (is\_sorted == false) {

for (int i = r; i > l; i--) {

if (A[i] < A[i - 1]) {

swap(A[i], A[i - 1]);

is\_sorted = false;

}

}

r--;

l++;

}

}