ВОПРОСЫ

по дисциплине «ОАиП»

за 1 семестр 2018-2019 г.(зима)

1. **Алгоритм. Классы алгоритмов. Способы записи алгоритмов. Свойства алгоритма.**

**Алгоритм** — это набор инструкций, описывающих порядок действий исполнителя для достижения поставленной цели за конечное число шагов.

**Свойства алгоритма:**

- дискретность (описанный процесс должен быть разбит на последовательность шагов, отделенных друг от друга, образующих дискретную структуру алгоритма)

- понятность (каждая инструкция должна быть понятна исполнителю)

- определенность и детерминированность (каждое правило алгоритма должно быть четким, однозначным)

- результативность (всегда должен быть результат. Отсутствие результата – тоже результат)

- массовость (должен быть применим для класса задач)

- конечность (результаты должны получаться за конечное число шагов и всегда должна быть точка выхода)

**Алгоритм считается правильным**, если его выполнение дает правильный результат.

Все алгоритмы могут быть разделены на:

* Линейные
* Разветвляющиеся
* циклические
* вспомогательные

Классы алгоритмов:

1. Вычислительные – исходные данные простые и их немного; процесс вычислений долгий и сложный. Матем. и физич. задачи.

2. Информационные – не очень сложные вычисления, объем данных большой. Требуют хорошей организации данных. Эконом. задачи.

3. Управляющие – исходные данные поступают от процесса или объекта, которым управляют. Результат – воздействие на процесс. Автоматизированное управление технолог. процессами (участие человека на определенных этапах; автоматическое – без участия).

4. Реального времени – результат должен быть получен к определенному времени.

**Способы описания алгоритма.**

*Способы записи алгоритма:*

**• словесный**: записи на естественном языке, описание словами последовательности выполнения алгоритма.

**• структурная схема алгоритма(графический)**: с помощью блок-схем.

Графический способ представления алгоритмов является более компактным и наглядным по сравнению со словесным. При графическом исполнении алгоритм изображается в виде последовательности связанных между собой блочных символов, каждый из которых соответствует выполнению одного из действий. Такое графическое представление называется схемой алгоритма или блок-схемой. В блок-схеме каждому типу действий соответствует геометрическая фигура, представленная в виде блочного символа. Блочные символы соединяются линиями переходов, определяющими очередность выполнения действий.

**• алгоритмический (псевдокод)**: компактный язык описания алгоритмов, использующий ключевые слова языков программирования, но опускающий несущественные подробности и специфический синтаксис; обычно опускает детали, несущественные для понимания алгоритма человеком. Главная цель — обеспечить понимание алгоритма человеком, сделать описание более воспринимаемым, чем исходный код на языке программирования.

**• программный код**

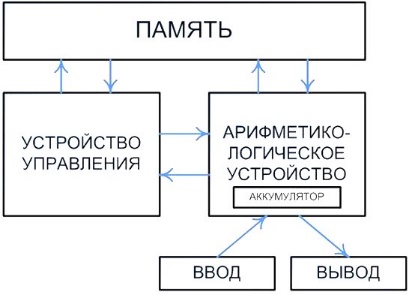
1. **Архитектура Фон – Неймана**

Архитектура фон Неймана — широко известный принцип совместного хранения [команд](https://ru.wikipedia.org/wiki/%252525D0%25252598%252525D0%252525BD%252525D1%25252581%252525D1%25252582%252525D1%25252580%252525D1%25252583%252525D0%252525BA%252525D1%25252586%252525D0%252525B8%252525D1%2525258F_(%252525D0%252525B8%252525D0%252525BD%252525D1%25252584%252525D0%252525BE%252525D1%25252580%252525D0%252525BC%252525D0%252525B0%252525D1%25252582%252525D0%252525B8%252525D0%252525BA%252525D0%252525B0)) и [данных](https://ru.wikipedia.org/wiki/%252525D0%25252594%252525D0%252525B0%252525D0%252525BD%252525D0%252525BD%252525D1%2525258B%252525D0%252525B5_(%252525D0%252525B2%252525D1%2525258B%252525D1%25252587%252525D0%252525B8%252525D1%25252581%252525D0%252525BB%252525D0%252525B8%252525D1%25252582%252525D0%252525B5%252525D0%252525BB%252525D1%2525258C%252525D0%252525BD%252525D0%252525B0%252525D1%2525258F_%252525D1%25252582%252525D0%252525B5%252525D1%25252585%252525D0%252525BD%252525D0%252525B8%252525D0%252525BA%252525D0%252525B0)) в [памяти компьютера](https://ru.wikipedia.org/wiki/%252525D0%2525259A%252525D0%252525BE%252525D0%252525BC%252525D0%252525BF%252525D1%2525258C%252525D1%2525258E%252525D1%25252582%252525D0%252525B5%252525D1%25252580%252525D0%252525BD%252525D0%252525B0%252525D1%2525258F_%252525D0%252525BF%252525D0%252525B0%252525D0%252525BC%252525D1%2525258F%252525D1%25252582%252525D1%2525258C). В общем случае, когда говорят об архитектуре фон Неймана, подразумевают принцип хранения данных и инструкций в одной памяти. В 1946 году Д. фон Нейман, Г. Голдстайн и А. Беркс в своей совместной статье изложили новые принципы построения и функционирования ЭВМ. В последствие на основе этих принципов производились первые два поколения компьютеров. В более поздних поколениях происходили некоторые изменения, хотя принципы Неймана актуальны и сегодня.

Принципы Фон-Неймана:

1. Использование двоичной системы счисления в вычислительных машинах.
2. Программное управление ЭВМ. Работа ЭВМ контролируется программой, состоящей из набора команд. Команды выполняются последовательно друг за другом.
3. Память компьютера используется не только для хранения данных, но и программ.
4. Ячейки памяти ЭВМ имеют адреса, которые последовательно пронумерованы. В любой момент можно обратиться к любой ячейке памяти по ее адресу.
5. Возможность условного перехода в процессе выполнения программы. Несмотря на то, что команды выполняются последовательно, в программах можно реализовать возможность перехода к любому участку кода.

Самым главным следствием этих принципов можно назвать то, что теперь программа уже не была постоянной частью машины (как например, у калькулятора). Программу стало возможно легко изменить.



1. Базовые конструкции структурированного программирования. Цель использования базовых конструкций.

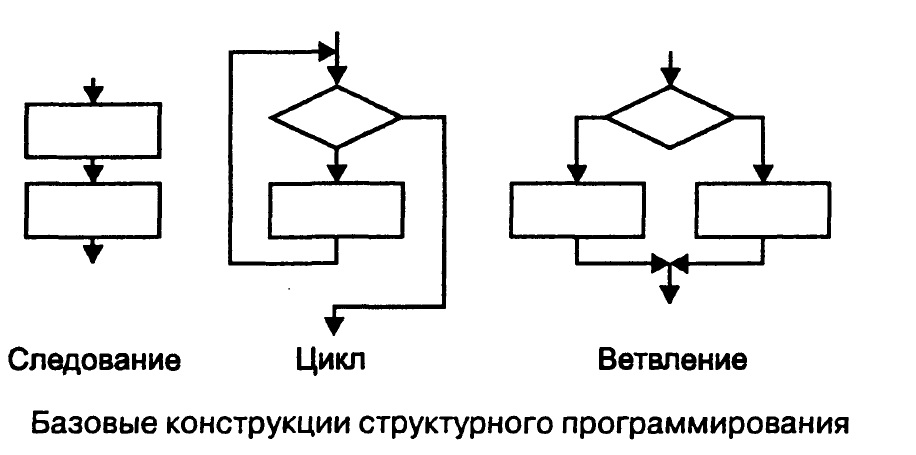
**Структурное программирование** — метод разработки по, в соответствии с которым программа представлена в виде структуры блоков, расположенных в определенном порядке.

Программу для решения задачи любой сложности можно составить только из трех структур: линейной, разветвляющейся и циклической. Эти структуры называются базовыми конструкциями структурного программирования.

Линейной конструкция (следование) – конструкция, представляющая собой последовательное соединение двух или более операторов.

Ветвление – задает выполнение одного из двух операторов, в зависимости от выполнения какого-либо условия.

Цикл – задает многократное выполнение оператора.



Цель использования базовых конструкций:

1. Обеспечить дисциплину программирования

2. Улучшить читабельность программы

3. Повысить эффективность программы

4. Повысить надежность программы

Структурное программирование часто называют «программирование без goto», и в этом есть большая доля правды: частое использование операторов передачи управления в произвольные точки программы затрудняет прослеживание логики ее работы. С другой стороны, никакие принципы нельзя возводить в абсолют, и есть ситуации, в которых использование goto оправдано и приводит, напротив, к упрощению структуры программы.

1. **Базовые средства языка СИ. Состав языка. Основные элементы алгоритмического языка.**

В алгоритмическом языке а*лфавит языка*, или его символы – это основные знаки, с помощью которых пишутся все тексты на языке. *Лексема*, или элементарная конструкция – минимальная единица языка, имеющая самостоятельный смысл. *Выражение* задает правило вычисления некоторого значения. *Оператор* задает законченное описание некоторого действия. Для описания сложного действия требуется последовательность операторов.

Объединенная единым алгоритмом совокупность описаний и операторов образует *программу*. Для выполнения программы ее необходимо перевести на язык понятный процессору – *в машинные коды*.

**Алфавит языка Си включает:**

**•** прописные и строчные буквы латинского алфавита, а также знак подчеркивания (код ASCII 95);

**•** арабские цифры от 0 до 9;

**•** специальные символы: +(плюс) –(минус) \*(звездочка) /(дробная черта) =(равно) >(больше) <(меньше) ;(точка с запятой) &(амперсант) [ ](квадратные скобки) { }(фигурные скобки) ()(круглые скобки) \_(знак подчеркивания) (пробел) .(точка) ,(запятая) :(двоеточие) #(номер) %(процент) ~(поразрядное отрицание) ?(знак вопроса) !(восклицательный знак) \(обратный слеш).

**•** пробельные (разделительные) символы: пробел, символы табуляции, перевода строки, возврата каретки, новая страница и новая строка.

Из символов алфавита формируются **лексемы языка** – минимальная единица языка, имеющая самостоятельный смысл:

1)идентификаторы (используется в качестве имени объекта, например, переменной, функции и т.п. Идентификатор может содержать до 32 символов и состоит из букв и цифр, но начинается обязательно с буквы. Строчные буквы отличаются от прописных. Идентификатор создается на этапе объявления переменной, функции, константы и т.п. и используется в последующих операторах программы);

2)ключевые (зарезервированные) слова (*Ключевые слова –* это зарезервированные слова, которые имеют специальное значение для компилятора. Их можно использовать только в том смысле, в котором они определены);

3)знаки операций;

4)константы;

5)разделители (скобки, точка, запятая, пробельные символы);

6) комментарии (*Комментарий* – это текст, который компилятором игнорируется. Начинается он либо с символов // и заканчивается переходом на новую строку либо заключается в скобки: /\* Это комментарий \*/)

**Границы лексем** определяются другими лексемами, такими, как разделители или знаки операций, а также комментариями.

**Основные элементы языка**:

1. Идентификаторы
2. Операции

* Арифметические
* Логические
* Отношения

1. Ключевые слова
2. Данные

* Константы
* Переменные
* Массивы

1. Выражения

* Арифметические
* Логические
* Строковые

1. Операторы
2. **Ввод\вывод строки.**

Строка в СИ – массив элементов типа char, в конце помещен символ '\0' (нуль-терминатор).

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. Строковая константа – это набор символов, заключенных в двойные кавычки. Строка при объявлении может быть инициализирована начальным значением, например, так:

char str[10]; //объявление строки

char str1[10] = {'Y','o','n','g','C','o','d','e','r','\0'}; //инициализация

char str2[10] = "Hello!";

char str3[] = "Hello!";

Если подсчитать кол-во символов в двойных кавычках после символа равно их окажется 9, а размер строки 10 символов, последнее место отводится под нуль–терминатор, причём компилятор сам добавит его в конец строки.

При объявлении строки не обязательно указывать её размер, но при этом обязательно нужно её инициализировать начальным значением. Тогда размер строки определится автоматически и в конец строки добавится нуль-терминатор. Строка может содержать символы, цифры и специальные знаки. В СИ строки заключаются в двойные кавычки. Имя строки является константным указателем на первый символ.

Для **ввода** строк обычно используются стандартные функции:

scanf() - вводит значения для строковых переменных спецификатором ввода %s до появления первого символа “пробел” (символ «&» перед ID строковых данных указывать не надо);

gets() - ввод строки с пробелами внутри этой строки завершается нажатием клавиши ENTER. Обе функции автоматически ставят в конец строки нулевой байт.

Функция fgets (str, 10, stdin) 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 (stdin – это стандартный поток ввода данных, обычно связанный с клавиатурой).

**Вывод** строк производится функциями printf() или puts() до первого нулевого байта („\0‟): printf() не переводит курсор после вывода на начало новой строки; puts() автоматически переводит курсор после вывода строковой информации в начало новой строки.

#include <stdio.h>

#include <string.h>

int main(void)

{

char str[80];

gets(str);

printf("Length is %d", strlen(str));

return 0;

}

Функция puts() выводит передаваемый ей аргумент на экран, завершая вывод переходом на новую строку. Она имеет следующий прототип:

*int puts(const char \*str);* Здесь str - это выводимая строка. Функция возвращает нецелое число в случае удачи и EOF - в случае неудачи.

fputs (std, stdout).

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

1. **Двумерный динамический массив. Выделение/высвобождение памяти.**

Двумерный массив – это массив, в котором каждый элемент является также массивом.

Двумерный массив целочисленных элементов можно инициализировать как: int s[3][5]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}. По аналогии с первым случаем это соответствует набору следующих операторов присвоения:

int s[0][0]=1; int s[0][1]=2; и т.д.

Вообще многомерные массивы, в том числе и двумерные, можно инициализировать как массив массивов. Например, две следующие инициализации равны между собой:

int s[3][5]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}; и

int s[3][5]={{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}};

Во втором случае ввели дополнительные фигурные скобки для записи элементов каждой строки двумерного массива. Если при этом количество элементов в строке не совпадает с числом столбцов в массиве, то соответствующие пропущенные элементы строки считаются неопределенными, так как они не определены.

При формировании двумерного динамического массива сначала выделяется память для массива указателей на одномерные массивы, а затем в цикле с параметром выделяется память под одномерные массивы.

* 1. int \*\*A = new int\*[N]; // созд-е массива указателей

for (int i = 0; i < N; i++) { //созд-е строк

A[i] = new int[M]; } //N – количество строк; M – количество столбцов

* 1. int \*\*A = (int \*\*)malloc(N\*sizeof(int \*));

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

A[i] = (int \*)malloc(M\*sizeof(int)); }

Удаление из динамической памяти двумерного массива осуществляется в порядке, обратном его созданию, то есть сначала высвобождается память, выделенная под одномерные массивы с данными, а затем память, выделенная под одномерные массив указателей:

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

delete[] A[i]; } //очищаем память

delete[] A;

1. for (int i=0; i < N; i++)

free(A[i]);

free(A);

**Пример программы:**

int main()

{

int m, n;

scanf("%d%d", &m, &n);

int \*\*A = NULL;

A = (int\*\*)calloc(m, sizeof(int));

for (int i=0; i < m; i++)

A[i] = (int\*)calloc(n, sizeof(int));

for (int m1=0, ch=1; m1<m; m1++) {

for (int n1=0; n1<n; n1++, ch++)

{ A[m1][n1] = ch;

cout << A[m1][n1];}

cout<<endl; }

for (int i=0; i<n; i++)

free(A[i]); //очистка памяти

free(A); //очистка памяти

return 0;

}

1. **Двумерный массив. Инициализация многомерного массива.**

Можно объявлять не только двумерные массивы, но и массивы с большим количеством измерений. Например, объявление int A[n][m][l] создает трехмерный массив из n\*m\*l элементов. Для обращения к каждому элементу такого массива необходимо указать три индекса: A[i][j][k], при этом 0<=i, i<n, 0<=j, j<m, 0<=k, k<l. Количество измерений в массиве может быть практически бесконечным (т.е. достаточным для решения любых практических задач).

Двумерный массив – это массив, в котором каждый элемент является также массивом. При формировании двумерного динамического массива сначала выделяется память для массива указателей на одномерные массивы, а затем в цикле с параметром выделяется память под одномерные массивы.

Объявление многомерного массива: <тип> < ID >[размер1][размер2]…

Многомерные массивы задаются указанием каждого измерения в квадратных скобках. Например, оператор int matr [6] [8]; задает описание матрицы из 6 строк и 8 столбцов. Нумерация строк и столбцов начинается с 0. При инициализации двумерного массива он представляется как массив из массивов, при этом каждый массив заключается в свои фигурные скобки, либо задается общий список в том порядке, в котором элементы располагаются в памяти:

int mas [] [2]= { {1,2}, {0,2}, {1,0}};

int mas [3][2]={1,2,0,2,1,0};

Наиболее быстро изменяется последний индекс элементов массива, поскольку многомерные массивы в языке Си размещаются в памяти компьютера в последовательности столбцов. int a[3][4] = {{1,2,0,0},{9,-2,4,1},{-7,0,0,0}}; Если в какой-то группе { } список значений короче, то оставшимся элементам присваивается 0.

Инициализация массива с помощью циклов, элементы генерируются рандомно:

for (int i = 0; i < N; i++) //N – количество строк; M – количество столбцов

{

for (int j = 0; j < M; j++) { //заполняем массив рандомными числами

A[i][j] = rand() % 50;

cout << A[i][j] << " "; }

cout << endl;

}

Пример программы:

#include <stdio.h>

int main()

{int a[2][3] = { 1, 2, 3, 4, 5, 6 };

printf("%d %d %d\n", a[0][0], a[0][1], a[0][2]);

printf("%d %d %d\n", a[1][0], a[1][1], a[1][2]);

getchar();

return 0;}

**Расположение двумерных массивов в памяти.**

Иногда, бывает полезно знать, как матрицы располагаются в памяти ЭВМ. Оказывается, во всех современных языках программирования (кроме Фортрана) элементы матрицы располагаются по строкам, то есть сначала изменяется последний индекс. Объявленная выше матрица X расположена так:

X[0][0] X[0][1] X[0][2] X[1][0] X[1][1] X[1][2]

1. Динамическая память функции malloc(), calloc(), realloc(), free()

Все переменные, объявленные в программе размещаются в одной непрерывной области памяти, которую называют сегментом данных. Такие переменные не меняют своего размера в ходе выполнения программы и называются статическими. Размера сегмента данных может быть недостаточно для размещения больших массивов информации. Выходом из этой ситуации является использование динамической памяти.

**Динамическая память** – это память, выделяемая программе для ее работы за вычетом сегмента данных, стека, в котором размещаются локальные переменные подпрограмм и собственно тела программы. Для работы с динамической памятью используют указатели. С их помощью осуществляется доступ к участкам динамической памяти, которые называются динамическими переменными. Динамические переменные создаются с помощью специальных функций и операций. Они существуют либо до конца работы программ, либо до тех пор, пока не будут уничтожены с помощью специальных функций или операций.

В языке Си определены библиотечные функции для работы с динамической памятью, они находятся в библиотеке :

1. **void\*malloc(unsigned s)** – возвращает указатель на начало области динамической памяти длиной s байт, при неудачном завершении возвращает NULL;

**void\*malloc(size\_t size);** Malloc выделяет size байт памяти и возвращает указатель на выделенную память. Если память выделить не удалось то она возвращает нулевой указатель. В качестве входного параметра функция принимает размер памяти, которую требуется выделить. Возвращаемым значением является указатель на выделенный в куче участок памяти.

1. **void\*calloc(unsigned n, unsigned m)** – возвращает указатель на начало области динамической для размещения n элементов длиной m байт каждый, при неудачном завершении возвращает NULL;

**void\*сalloc(size\_f num, size\_t size);** выделяет память количества элементов num, размером size:num\*size байт. Функция работает аналогично malloc, но отличается синтаксисом (вместо размера выделяемой памяти нужно задать количество элементов и размер одного элемента) и тем, что выделенная память будет обнулена.

1. **void\*realloc(void \*p,unsigned s)** – изменяет размер блока ранее выделенной динамической до размера s байт, р – адрес начала изменяемого блока, при неудачном завершении возвращает NULL;

**void\*realloc(void\*ptr, size\_t size);** realloc позволяет изменить размер выделенной памяти по указателю ptr на новый размер size

Если существует возможность изменения размера для текущего указателя ptr, то будет выполнено перераспределение по этому указателю, иначе будет выделена новая память о с сохранением всех данных

1. **void \*free(void \*p)** – освобождает ранее выделенный участок динамической памяти, р- адрес начала участка.

**free(void\*ptr)** выполняет освобождение памяти . Функция изменяет размер выделенной памяти (на которую указывает ptr, полученный из вызова malloc, calloc или realloc). Если размер указанный в параметре size больше, чем тот, который был выделен под указатель ptr, то проверяется, есть ли возможность выделить недостающие ячейки памяти подряд с уже выделенными. Если места недостаточно, то выделяется новый участок памяти размером size и данные по указателю ptr копируются в начало нового участка.

Пример:

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

free(u); //освобождение выделенной памяти в функцию передается количество требуемой памяти в байтах, т. к. функция возвращает значение типа void\*, то его необходимо преобразовать к типу указателя (int\*).

1. **Доступ к элементам массива. Действия над элементами массива.**

Массив – структурированный тип данных с элементами одного и того же типа, имеющий одно имя и определенное количество элементов, расположенных в памяти строго последовательно. Количество элементов определяет размер массива. Порядковый номер элемента массива называется его индексом. Число индексов называется размерностью массива, например, массив с двумя индексами называется двумерным массивом. Вектор – одномерный массив элементов, матрица – массив векторов.

Обработка массивов выполняется обычно следующим образом: объявление, ввод или инициализация элементов массива, преобразование и вывод.

Обращение к элементу массива

Каждый элемент массива имеет свой порядковый номер. Чтобы обратиться к элементу массива, надо написать имя массива и затем в квадратных скобках номер нужного элемента. Важно запомнить правило: элементы массивов в языке Си нумеруются с нуля. Поэтому индекс последнего элемента массива на 1 меньше числа элементов в данном массиве.

Таким образом, если в массиве 10 элементов, он содержит элементы:

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

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

x = (A[3] + 5)\*A[1]; // прочитать значения A[3] и A[1]

A[0] = x + 6; // записать новое значение в A[0]

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

Действия над элементами массива:

1)Ввод с клавиатуры и вывод на экран

Существует много способов ввода в зависимости от вашей задачи:

• элементы массива вводятся с клавиатуры вручную;

• массив заполняется случайными числами (например, для моделирования

случайных процессов);

• элементы массива читаются из файла;

• массив заполняется в процессе вычислений.

Чтобы ввести массив в память, надо каждый его элемент обработать отдельно (например, вызвав для него функцию ввода scanf). Для ввода и вывода массива обычно используется цикл for.

2)Индексированные элементы массива называются индексированными переменными и могут быть использованы так же, как и простые переменные. Например, они могут находиться в выражениях в качестве операндов, использоваться в операторах, им можно присваивать любые значения, соответствующие их типу.

3)Копированием массивов называется присваивание значений всех элементов одного массива всем соответствующим элементам другого массива.

4)Поиск в массиве элементов, удовлетворяющих некоторым условиям

5)Перестановка значений элементов массива осуществляется с помощью дополнительной переменной того же типа, что и базовый тип массива.

6)**Sizeof(<параметр>):** принимает один параметр и выражает размер пямяти в байтах указанного объекта в качестве параметра. Параметром модель быть идентификатор некоторой переменной или тип данных. Операция если принимает в качестве оператора идентификатор массива, то возвращает размер массива умноженный на размер одного Элемента в памяти.

1. **Идентификаторы. Ключевые слова. Константы**

**Идентификатор** (ID) – это имена которыми обозначаются некоторые объекты в программе. В идентификаторе могут использоваться латинские буквы, цифры и знак подчеркивания; первым символом ID может быть буква или знак подчеркивания, но не цифра; пробелы внутри ID не допускаются. Прописные и строчные буквы различаются, например, respo, ReSpO и RESPO — три различных имени. Длина идентификатора определяется реализацией (версией) транслятора Cи и редактора связей (компоновщика). Разделители идентификаторов объектов:

- пробелы;

- символы табуляции, перевода строки и страницы;

- комментарии (играют роль пробелов).

Наличие разделителей не влияет на работу программы. В Си прописные и строчные буквы – различные символы.

Идентификатор – составная часть языка, имеют фиксированное написание и определенный смысл и не используются для других целей.

При выборе идентификатора учитываем:

1. не должен совпадать с ключевыми словами языка и именами функций
2. не рекомендуется начинать со знака «\_», потому что именам транслируемых функций компилятор автоматически добавляет знак подчеркивания.
3. правильно выбранный идентификатор облегчает чтение и выбор понимания программы.

Идентификатор создается на этапе объявления переменной, функции, типа и т. п., после этого его можно использовать в последующих операторах программы.

**Ключевые слова** — это зарезервированные идентификаторы, которые имеют специальное значение для компилятора. Они сообщают компилятору о типе данных, способе организации и порядке выполнения. Их можно использовать только в том смысле, в котором они определены. Например: for, if, do, int, double, while, auto, struct, break, case, char, const, continue, default, else, float, goto, long, return, short, signed, sizeof, switch, unsigned, void.

**Константы** — это данные, не изменяющие своего значения в ходе выполнения программы. В языке СИ можно выделить четыре вида констант: целые константы, константы с плавающей запятой, символьные константы и строковые литералы.

Когда в программе встречается некоторое число, например 1, то это число называется **литералом или литеральной константой**. Константой, потому что мы не можем изменить его значение, и литералом, потому что буквально передает свое значение (от латинского literal – буквальный). Литерал является не адресуемой величиной: хотя реально он, конечно, хранится в памяти машины, нет никакого способа узнать его адрес. Каждый литерал имеет определенный тип.

1. **Классификация языков программирования.**

Существуют различные классификации языков программирования. Если язык близок к естественному языку программирования, то он называется языком высокого уровня, если ближе к машинным командам, – языком низкого уровня.

В группу языков **низкого** уровня входят машинные языки и языки символического кодирования (пример: Ассемблер). Операторы этого языка – это те же машинные команды, но записанные мнемоническими кодами, а в качестве операндов используются не конкретные адреса, а символические имена. Все языки низкого уровня ориентированы на определенный тип компьютера, т. е. являются машинно-зависимыми.

Машинно-ориентированные языки – это языки, наборы операторов и изобразительные средства которых существенно зависят от особенностей ЭВМ (внутреннего языка, структуры памяти и т.д.).

К языкам программирования **высокого** уровня относят Паскаль, Бейсик, Си и другие. Эти языки машинно-независимы, т.к. они ориентированы не на систему команд той или иной ЭВМ, а на систему операндов, характерных для записи определенного класса алгоритмов. Однако программы, написанные на языках высокого уровня, занимают больше памяти и медленнее выполняются, чем программы на машинных языках.

Программу, написанную на языке программирования высокого уровня, ЭВМ не понимает, поскольку ей доступен только машинный язык. Поэтому для перевода программы с языка программирования на язык машинных кодов используют специальные программы – трансляторы. Самый яркий пример транслятора: компиляторы- преобразует всю программу в модуль на машинном языке, после чего программа записывается в память компьютера и лишь потом исполняется.

Языки программирования также можно разделять на поколения:

– **языки первого поколения**: машинно-ориентированные с ручным управлением памяти на компьютерах первого поколения.

– **языки второго поколения**: с мнемоническим представлением команд, так называемые автокоды.

– **языки третьего поколения**: общего назначения, используемые для создания прикладных программ любого типа. Например, Бейсик, Си и Паскаль.

– **языки четвертого поколения**: усовершенствованные, разработанные для создания специальных прикладных программ, для управления базами данных.

– **языки программирования пятого поколения**: языки декларативные, объектно–ориентированные и визуальные. Например, Пролог, Си++, Visual Basic, Delphi.

**По структурному строению программ:**

**•** *процедурно-ориентированные языки (Pascal, C и др.)* – используется метод разбивки всей задачи на более простые подзадачи (процедуры или функции) и их независимая реализация. Достоинство - создание достаточно сложных программ.

**•** *объектно-ориентированные языки (C++, Java и др.)* – из области решаемой проблемы выделяются классы, объединяющие связанные каким-либо признаком данные и функции по их обработке. Далее создаются объекты данных классов, которые, взаимодействуя друг с другом, осуществляют решение поставленной задачи.

1. **Массивы. Объявления. Инициализация.**

Массив – структурированный тип данных с элементами одного и того же типа, имеющий одно имя и определенное количество элементов, расположенных в памяти строго последовательно. Количество элементов определяет размер массива. Порядковый номер элемента массива называется его индексом. Число индексов называется размерностью массива, например, массив с двумя индексами называется двумерным массивом. Вектор – одномерный массив элементов, матрица – массив векторов.

Обработка массивов выполняется обычно следующим образом: объявление, ввод или инициализация элементов массива, преобразование и вывод.

Объявление массива

Объявить массив – выделить место в памяти компьютера, объём которой зависит от количества элементов и типа массива. Тип массива - это тип входящих в него элементов. Массивы могут быть разных типов: int, float, char, и т.д. Массив объявляют так же, как и обычные переменные, но после имени массива в квадратных скобках записывается его размер.

int A[10], B[20]; // 2 массива по 10 и 20 целых чисел

float C[12]; // массив из 12 вещественных чисел

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

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

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

int A[4] = { 2 }; // последние три элемента равны 0  
int A[4] = { 2, 3, 12, 76, 0 }; // ошибка, должно быть не более 4-х элементов

Для повышения универсальности программы размер массива лучше определять через константу. В этом случае для переделки программы для массива другого размера надо только поменять значение этой константы:

const int N = 20; //N - константа  
main()  
{ int A[N]; // размер массива задан через константу ...

}

Двумерный массив – это массив, в котором каждый элемент является также массивом. При формировании двумерного динамического массива сначала выделяется память для массива указателей на одномерные массивы, а затем в цикле с параметром выделяется память под одномерные массивы.

Объявление многомерного массива: <тип> < ID >[размер1][размер2]…

Многомерные массивы задаются указанием каждого измерения в квадратных скобках. Например, оператор int matr [6] [8]; задает описание матрицы из 6 строк и 8 столбцов. Нумерация строк и столбцов начинается с 0. При инициализации двумерного массива он представляется как массив из массивов, при этом каждый массив заключается в свои фигурные скобки, либо задается общий список в том порядке, в котором элементы располагаются в памяти:

int mas [] [2]= { {1,2}, {0,2}, {1,0}};

int mas [3][2]={1,2,0,2,1,0};

Наиболее быстро изменяется последний индекс элементов массива, поскольку многомерные массивы в языке Си размещаются в памяти компьютера в последовательности столбцов. int a[3][4] = {{1,2,0,0},{9,-2,4,1},{-7,0,0,0}}; Если в какой-то группе { } список значений короче, то оставшимся элементам присваивается 0.

Инициализация массива с помощью циклов, элементы генерируются рандомно:

for (int i = 0; i < N; i++) //N – количество строк; M – количество столбцов

{

for (int j = 0; j < M; j++) { //заполняем массив рандомными числами

A[i][j] = rand() % 50;

cout << A[i][j] << " "; }

cout << endl; }

1. **Массивы. Объявления. Работа с указателями.**

Массив - структурированный тип данных с элементами одного и того же типа, имеющий одно имя и определенное количество элементов, расположенных в памяти строго последовательно. Количество элементов определяет размер массива. Порядковый номер элемента массива называется его индексом. Число индексов называется размерностью массива, например, массив с двумя индексами называется двумерным массивом. Вектор – одномерный массив элементов, матрица – массив векторов.

Обработка массивов выполняется обычно следующим образом: объявление, ввод или инициализация элементов массива, преобразование и вывод.

**Объявление массива**

Объявить массив – выделить место в памяти компьютера, объём которой зависит от количества элементов и типа массива. Тип массива - это тип входящих в него элементов. Массивы могут быть разных типов: int, float, char, и т.д. Массив объявляют так же, как и обычные переменные, но после имени массива в квадратных скобках записывается его размер.

int A[10], B[20]; // 2 массива по 10 и 20 целых чисел

float C[12]; // массив из 12 вещественных чисел

Для повышения универсальности программы размер массива лучше определять через константу. В этом случае для переделки программы для массива другого размера надо только поменять значение этой константы:

const int N = 20; //N - константа  
main(){

int A[N]; } // размер массива задан через константу ...

Объявление многомерного массива: <тип> < ID >[размер1][размер2]…

Многомерные массивы задаются указанием каждого измерения в квадратных скобках. Например, оператор int matr [6] [8]; задает описание матрицы из 6 строк и 8 столбцов. Нумерация строк и столбцов начинается с 0. При инициализации двумерного массива он представляется как массив из массивов, при этом каждый массив заключается в свои фигурные скобки, либо задается общий список в том порядке, в котором элементы располагаются в памяти:

int mas [] [2]= { {1,2}, {0,2}, {1,0}};

int mas [3][2]={1,2,0,2,1,0};

Наиболее быстро изменяется последний индекс элементов массива, поскольку многомерные массивы в языке Си размещаются в памяти компьютера в последовательности столбцов. int a[3][4] = {{1,2,0,0},{9,-2,4,1},{-7,0,0,0}}; Если в какой-то группе { } список значений короче, то оставшимся элементам присваивается 0.

Важно запомнить правило: элементы массивов в языке Си нумеруются с нуля. Поэтому индекс последнего элемента массива на 1 меньше числа элементов в данном массиве.

Номер элемента массива также называется его индексом. Вот примеры обращения к массиву A:  
x = (A[3] + 5)\*A[1]; // прочитать значения A[3] и A[1]

A[0] = x + 6; // записать новое значение в A[0].

**Работа с указателями**

С указателями можно выполнять следующие операции:

* разыменование (\*);
* присваивание;
* сравнение;
* сложение, вычитание, арифметические инкремент ++, декремент --
* приведение типов.

Операция разыменования предназначена для получения значения переменной или константы, адрес которой хранится в указателе. Если указатель указывает на переменную, то это значение можно изменять, также используя операцию разыменования.

int a; //переменная типа int

int\* pa=new int; //указатель и выделение памяти под динамическую переменную

\*pa=10; //присвоили значение динамической переменной, на которую указывает

указатель

a=\*pa; //присвоили значение переменной а

Арифметические операции применимы только к указателям одного типа:

1) Инкремент увеличивает значение указателя на величину sizeof(тип).

char\* pc;

int\* pi;

double\* pd;

. . . . .

pc++; //значение увеличится на 1

pi++; //значение увеличится на 4

pd++; //значение увеличится на 8

2) Декремент уменьшает значение указателя на величину sizeof(тип)

3) Разность двух указателей – это разность их значений, деленная на размер типа в байтах.

4) Суммирование двух указателей не допускается.

5) Можно суммировать указатель и константу.

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).

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

ptr1=a+5;

ptr2=a+7;

if (prt1>ptr2) a[3]=4;

В данном примере значение ptr1 меньше значения ptr2 и поэтому оператор a[3]=4 не будет выполнен.

1. **Одномерный массив. N-мерный массив.**

**Одномерные массивы в СИ.**

Одномерный массив — массив, с одним параметром, характеризующим количество элементов одномерного массива. Фактически одномерный массив — это массив, у которого может быть только одна строка, и n-е количество столбцов. Столбцы в одномерном массиве — это элементы массива. Нумерация ячеек массива всегда начинается с 0. Индекс ячейки – это целое неотрицательное число, по которому можно обращаться к каждой ячейке массива и выполнять какие-либо действия над ней (ячейкой).

#include <stdio.h>

int main()

{

int a[] = { 5, 4, 3, 2, 1 }; // массив a содержит 5 элементов

printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);

getchar();

return 0;

}

Индексы у одномерных массивов в языке Си начинаются с 0, а в программе одномерный массив объявляется следующим образом:

<тип> < ID\_массива>[размер]={список начальных значений};

где: тип – базовый тип элементов (целый, вещественный, символьный); размер – количество элементов массива. Список начальных значений используется при необходимости инициализировать данные при объявлении, он может отсутствовать. Размер массива может задаваться константой или константным выражением. Нельзя задавать массив переменного размера. Для этого существует отдельный механизм – динамическое выделение памяти. Пример объявления массива целого типа: int a[5]; В массиве «а» первый элемент: а[0], второй – а[1], … пятый - а[4].

**Многомерные массивы**

Можно объявлять не только двумерные массивы, но и массивы с большим количеством измерений. Например, объявление int A[n][m][l] создает трехмерный массив из n\*m\*l элементов. Для обращения к каждому элементу такого массива необходимо указать три индекса: A[i][j][k], при этом 0<=i, i<n, 0<=j, j<m, 0<=k, k<l. Количество измерений в массиве может быть практически бесконечным (т.е. достаточным для решения любых практических задач).

Двумерный массив – это массив, в котором каждый элемент является также массивом. При формировании двумерного динамического массива сначала выделяется память для массива указателей на одномерные массивы, а затем в цикле с параметром выделяется память под одномерные массивы.

Объявление многомерного массива: <тип> < ID >[размер1][размер2]…

Многомерные массивы задаются указанием каждого измерения в квадратных скобках. Например, оператор int matr [6] [8]; задает описание матрицы из 6 строк и 8 столбцов. Нумерация строк и столбцов начинается с 0. При инициализации двумерного массива он представляется как массив из массивов, при этом каждый массив заключается в свои фигурные скобки, либо задается общий список в том порядке, в котором элементы располагаются в памяти:

int mas [] [2]= { {1,2}, {0,2}, {1,0}};

int mas [3][2]={1,2,0,2,1,0};

Наиболее быстро изменяется последний индекс элементов массива, поскольку многомерные массивы в языке Си размещаются в памяти компьютера в последовательности столбцов. int a[3][4] = {{1,2,0,0},{9,-2,4,1},{-7,0,0,0}}; Если в какой-то группе { } список значений короче, то оставшимся элементам присваивается 0.

#include <stdio.h>

int main()

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

printf("%d %d %d\n", a[0][0], a[0][1], a[0][2]);

printf("%d %d %d\n", a[1][0], a[1][1], a[1][2]);

getchar();

return 0;

}

**Расположение двумерных массивов в памяти.**

Иногда, бывает полезно знать, как матрицы располагаются в памяти ЭВМ. Оказывается, во всех современных языках программирования (кроме Фортрана) элементы матрицы располагаются по строкам, то есть сначала изменяется последний индекс. Объявленная выше матрица X расположена так: X[0][0] X[0][1] X[0][2] X[1][0] X[1][1] X[1][2].

Удаление из динамической памяти двумерного массива осуществляется в порядке, обратном его созданию, то есть сначала высвобождается память, выделенная под одномерные массивы с данными, а затем память, выделенная под одномерные массив указателей:

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

1. **Операторы цикла. Цель использования операторов цикла. Итерация. Параметры цикла.**

**Цикл** - организованное повторение некоторой последовательности операторов. Любой цикл обязательно содержит условие прерывания. Операторы, которые повторяются в цикле- тело цикла.

Перечень разновидностей операторов цикла:

- оператор цикла с предусловием;

- оператор цикла с постусловием;

- оператор цикла с предусловием и коррекцией.

Для организации циклов используются специальные операторы. Любой цикл состоит из кода цикла, т.е. тех операторов, которые выполняются несколько раз, начальных установок, модификации параметра цикла и проверки условия продолжения выполнения цикла. Проверка условия выполняется на каждой итерации либо до кода цикла (с предусловием), либо после кода цикла (с постусловием).Любой цикл состоит из тела цикла, то есть тех операторов, которые выполняются несколько раз, начальных установок, модификации параметра цикла и проверки условия продолжения выполнения цикла.

**Параметр цикла** - переменная, управляющая выполнением цикла. Параметр цикла изменяет свое значение при каждом проходе тела цикла.

**Один проход цикла называется итерацией**. Проверка условия выполняется на каждой итерации либо до тела цикла (тогда говорят о цикле с предусловием), либо после тела цикла (цикл с постусловием). Разница между ними состоит в том, что тело цикла с постусловием всегда выполняется хотя бы один раз, после чего проверяется, надо ли его выполнять еще раз. Проверка необходимости выполнения цикла с предусловием делается до тела цикла, поэтому возможно, что он не выполнится ни разу.

Операторы цикла взаимозаменяемы, но можно привести некоторые рекомендации по выбору наилучшего в каждом конкретном случае.

Оператор do while обычно используют, когда цикл требуется обязательно выполнить хотя бы раз (например, если в цикле производится ввод данных).

Оператором while удобнее пользоваться в случаях, когда число итераций заранее не известно, очевидных параметров цикла нет или модификацию параметров удобнее записывать не в конце тела цикла.

Оператор for предпочтительнее в большинстве остальных случаев (однозначно — для организации циклов со счетчиками).

1. **Операторы. Оператор перехода**

**Оператор goto**

Оператор безусловного перехода goto имеет формат: goto метка;

В теле той же функции должна присутствовать ровно одна конструкция вида:

метка: оператор;

int n=m=10; int=a;

for(int :=0; i<n; i++)

{ for(in j=0; j<m; j++)

{ scanf(‘’%d’’, &a);

if(a<0) GOTO Error;} } //обработка а

Error: printf(‘’Ошибка‘’);

Оператор goto передает управление на помеченный оператор. Метка — это обычный идентификатор, областью видимости которого является функция, в теле которой он задан.

Использование оператора безусловного перехода оправдано в двух случаях:

* принудительный выход вниз по тексту программы из нескольких вложенных циклов или переключателей;
* переход из нескольких мест функции в одно (например, если перед выходом из функции всегда необходимо выполнять какие-либо действия).

В остальных случаях для записи любого алгоритма существуют более подходящие средства, а использование goto приводит только к усложнению структуры программы и затруднению отладки. Применение goto нарушает принципы структурного и модульного программирования, по которым все блоки, из которых состоит программа, должны иметь только один вход и один выход.

В любом случае не следует передавать управление внутрь операторов if, switch и циклов. Нельзя переходить внутрь блоков, содержащих инициализацию переменных, на операторы, расположенные после нее, поскольку в этом случае инициализация не будет выполнена.

**Оператор return** производит досрочный выход из текущей функции. Он, так же возвращает значение результата функции: return <выражение>;

#include <stdio.h>

int main()

{ printf("Hello world!**\n**");

return 0; }

В функциях, не возвращающих результат, он неявно присутствует после последнего оператора. Значение выражения при необходимости преобразуется к типу возвращаемого функцией значения. Вид оператора: return [ выражение ]; Операторов return может быть бесконечно много. Если функция возвращает какие-то значения, то в ней должен быть хоть один return.

**Оператор break** производит прерывание целого цикла или оператора-переключателя switch, к которому он принадлежит, и передает управление первому оператору, следующему за текущим оператором. Выполняет завершение работы функции и при необходимости возврат значений. Все операторы после Return не выполняются. Break используется для прекращения выполнения цикла из-за обнаружения ошибки, для организации дополнения к условию в заголовке цикла, для прекращения бесконечного цикла.

int st;  
scanf("%d",&st);  
while (st>0 && st<25)  
{ if st==4||st==8||st==12)  
break; }

**Оператор continue (**используется только в операторах цикла)

Оператор перехода к следующей итерации цикла continue пропускает все операторы, оставшиеся до конца тела цикла, и передает управление на начало следующей итерации. **Oператор continue** может использоваться во всех типах циклов, но не в операторах переключателя switch. Наличие оператора continue вызывает пропуск "оставшейся" части итерации и переход к началу следующей, т.е. досрочное завершение текущего шага и переход к следующему шагу. В циклах while и do это означает непосредственный переход к проверочной части. Для цикла for переход выполняется на выражение коррекции.

Оператор continue часто используется, когда последующая часть цикла оказывается слишком сложной, так что рассмотрение условия, обратного проверяемому, приводит к слишком высокому уровню вложенности программы.

1. **Операторы. Оператор условия.**

**Оператор if:**

*if(выражение){*

*блок операций 1;*

*} else {*

*блок операций 2;*

*}*

Если выражение истинно, то выполняется блок операций 1, иначе выполняется блок операций 2. Блок операций может состоять из одной операции. Ветвь else может отсутствовать, тогда при неверном выражении будет выполняться оператор, следующий за if. Оператор if может быть вложенным. При использовании вложенной формы оператора if опция else связывается с последним оператором if. Если требуется связать опцию else с предыдущим оператором if, внутренний условный оператор заключается в фигурные скобки.

**Условная операция «?»:** Условная операция - тернарная, в ней участвуют три операнда. Формат написания условной операции следующий:

**Условие? Оператор 1 : оператор 2;**

если условие истинно, то результатом операции является оператор 1, в противном случае - результатом операции является оператор 2. Каждый раз вычисляется только одно из выражений 1 или 2.

Запишем оператор if, вычисляющий максимум из а и b и присваивающий его значение z. if (a > b) z=a; else z=b; Используя условную операцию, этот пример можно записать: z = (a>b) ? a : b;

Условную операцию можно использовать так же, как и любое другое выражение. Если выражения 2 и 3 имеют разные типы, то тип результата определяется по правилам преобразования.

**Оператор switch:**

switch (целое выражение) {

case константа1: оператор1; break;

case константа2: оператор2; break;

…

case константа n: оператор n; break;

default: блок операций по умолчанию; break; }

Значение вычисленного выражения должно быть целого типа (или символьного, поскольку он автоматически преобразуется в целый).

Оператор break осуществляет выход из оператора switch и переход к следующему за ним оператору. При отсутствии опции break будут выполняться все операторы, начиная с помеченного данной меткой и кончая оператором в опции default. Альтернатива default может отсутствовать, тогда не будет произведено никаких действий. Константы в опциях case должны быть целого типа (могут быть символами). В случае несовпадения значения выражения с одной из констант происходит переход на метку default, либо, при ее отсутствии, к оператору, следующему за оператором switch.

Char ch=getchar();

Switch(ch) {

Case ‘a’: puts(‘’A’’); break;

Case ‘b’: puts(‘’B’’); break;

Case ‘c’: puts(‘’C’’); break; }

Оператор ветвления switch выполняется следующим образом:

* вычисляется целочисленное выражение в скобках оператора switch;
* полученное значение сравнивается с метками (константами) в опциях case, сравнение производится до тех пор, пока не будет найдена метка, соответствующая вычисленному значению целочисленного выражения;
* выполняется оператор соответствующей метки case;
* если соответствующая метка не найдена, то выполнится оператор в опции default.

1. **Операторы. Операторы безусловного перехода.**

**Оператор goto**

Оператор безусловного перехода goto имеет формат: goto метка;

В теле той же функции должна присутствовать ровно одна конструкция вида:

метка: оператор;

int n=m=10; int=a;

for(int :=0; i<n; i++)

{ for(in j=0; j<m; j++)

{ scanf(‘’%d’’, &a);

if(a<0) GOTO Error;} } //обработка а

Error: printf(‘’Ошибка‘’);

Оператор goto передает управление на помеченный оператор. Метка — это обычный идентификатор, областью видимости которого является функция, в теле которой он задан.

Использование оператора безусловного перехода оправдано в двух случаях:

* принудительный выход вниз по тексту программы из нескольких вложенных циклов или переключателей;
* переход из нескольких мест функции в одно (например, если перед выходом из функции всегда необходимо выполнять какие-либо действия).

В остальных случаях для записи любого алгоритма существуют более подходящие средства, а использование goto приводит только к усложнению структуры программы и затруднению отладки. Применение goto нарушает принципы структурного и модульного программирования, по которым все блоки, из которых состоит программа, должны иметь только один вход и один выход.

В любом случае не следует передавать управление внутрь операторов if, switch и циклов. Нельзя переходить внутрь блоков, содержащих инициализацию переменных, на операторы, расположенные после нее, поскольку в этом случае инициализация не будет выполнена.

1. **Операторы. Синтаксис. Семантика**

**Синтаксис** - это правила построения элементов языка.

**Семантика** определяет смысл и правила использования элементов языка.

**Простые операторы**:

- оператор присваивания

- оператор вызова функции

- пустой оператор «;»

Оператор задаёт законченное описание некоторых действий. Операторы могут быть объединены в составной оператор.

**Составной оператор** - это последовательность операторов, заключены в фигурные скобки.

* 1. **Операторы перехода:**

**Оператор break**

Оператор break используется внутри операторов цикла или switch для обеспечения перехода в точку программы, находящуюся непосредственно за оператором, внутри которого находится break.

**Оператор continue**

Оператор перехода к следующей итерации цикла continue пропускает все операторы, оставшиеся до конца тела цикла, и передает управление на начало следующей итерации.

**Оператор return**

Оператор возврата из функции return завершает выполнение функции и передает управление в точку ее вызова. Вид оператора:

return <выражение>;

**Оператор goto**

Оператор безусловного перехода goto имеет формат: **goto метка;**

В теле той же функции должна присутствовать ровно одна конструкция вида:

**метка: оператор;**

if(a<0) GOTO Error;

Error: printf(‘’Ошибка‘’);

Оператор goto передает управление на помеченный оператор. Метка — это обычный идентификатор, областью видимости которого является функция, в теле которой он задан.

Использование оператора безусловного перехода оправдано в двух случаях:

* принудительный выход вниз по тексту программы из нескольких вложенных циклов или переключателей;
* переход из нескольких мест функции в одно (например, если перед выходом из функции всегда необходимо выполнять какие-либо действия).

В остальных случаях для записи любого алгоритма существуют более подходящие средства, а использование goto приводит только к усложнению структуры программы и затруднению отладки.

В любом случае не следует передавать управление внутрь операторов if, switch и циклов. Нельзя переходить внутрь блоков, содержащих инициализацию переменных, на операторы, расположенные после нее, поскольку в этом случае инициализация не будет выполнена.

* 1. **Условные операторы:**

**Оператор if:**

*if(выражение){*

*блок операций 1;*

*} else {*

*блок операций 2;*

*}*

Если выражение истинно, то выполняется блок операций 1, иначе выполняется блок операций 2. Блок операций может состоять из одной операции. Ветвь else может отсутствовать, тогда при неверном выражении будет выполняться оператор, следующий за if. Оператор if может быть вложенным. При использовании вложенной формы оператора if опция else связывается с последним оператором if. Если требуется связать опцию else с предыдущим оператором if, внутренний условный оператор заключается в фигурные скобки.

**Оператор switch:**

switch (целое выражение) {

case константа1: блок операций1; break;

case константа2: блок операций2; break;

. . .

case константаn: блок операцийn; break;

default: блок операций по умолчанию; break;

}

Оператор ветвления switch выполняется следующим образом:

* вычисляется целочисленное выражение в скобках оператора switch;
* полученное значение сравнивается с метками (константами) в опциях case, сравнение производится до тех пор, пока не будет найдена метка, соответствующая вычисленному значению целочисленного выражения;
* выполняется оператор соответствующей метки case;
* если соответствующая метка не найдена, то выполнится оператор в опции default.

3. **Операторы цикла**

**Оператор do while** (цикл с постусловием) обычно используют, когда цикл требуется обязательно выполнить хотя бы раз (например, если в цикле производится ввод данных).

**do** *<оператор>* **while** *<выражение>*;

**Оператором while** (цикл с предусловием)удобнее пользоваться в случаях, когда число итераций заранее не известно, очевидных параметров цикла нет или модификацию параметров удобнее записывать не в конце тела цикла.

**while** (*<выражение>*) *<оператор>*

**Оператор for** (цикл с параметром) предпочтительнее в большинстве остальных случаев (однозначно — для организации циклов со счетчиками).

**for** (*<начальное выражение>*; *<условное выражение>*; *<выражение приращения>*)

{*<оператор>;}*

1. **Операции сдвига. Операции отношения.**

**Операции языка Си** предназначены для управления данными (более 40). Для их использования необходимо знать: - синтаксис; - приоритеты; - порядок выполнения. Выражения состоят из операндов, операций, скобок и используются для вычисления некоторого значения определенного типа. Каждый операнд может быть, в свою очередь, выражением или одним из его частных случаев.

**Операции сдвига ( << и >> ):**

Применяются к целочисленным операндам. Они сдвигают двоичное представление первого операнда влево или вправо на количество двоичных разрядов, заданное вторым операндом. При сдвиге влево (<<) освободившиеся разряды обнуляются (справа). При сдвиге вправо (>>) освободившиеся биты (слева) заполняются нулями, если первый операнд беззнакового типа, и знаковым разрядом в противном случае.

**Операции отношения (<. <=, >, >=, ==, !=):**

Сравнивают первый операнд со вторым. Операнды могут быть арифметического типа или указателями. Результатом операции является значение true или false (любое значение, не равное нулю, интерпретируется как true). Операции сравнения на равенство и неравенство имеют меньший приоритет, чем остальные операции сравнения.

1. **Операции увеличения и уменьшения на 1. Операция определения размера. Операции отрицания. Операция деления и остаток от деления.**

**Операции языка Си** предназначены для управления данными (более 40). Для их использования необходимо знать: - синтаксис; - приоритеты; - порядок выполнения. Выражения состоят из операндов, операций, скобок и используются для вычисления некоторого значения определенного типа. Каждый операнд может быть, в свою очередь, выражением или одним из его частных случаев.

**Операции увеличения и уменьшения на 1 (++ и** --**) :**

Эти операции, называемые также инкрементом и декрементом, имеют две формы записи — префиксную, когда операция записывается перед операндом, и постфиксную. В префиксной форме сначала изменяется операнд, а затем его значение становится результирующим значением выражения, а в постфиксной форме значением выражения является исходное значение операнда, после чего он изменяется.

**Операция определения размера sizeof :**

Она предназначена для вычисления размера объекта или типа в байтах, и имеет две формы:

sizeof выражение;

sizeof (тип);

**Операция отрицания (!):**

Логическое отрицание (!) дает в результате значение 0, если операнд есть истина (не нуль), и значение 1, если операнд равен нулю. Операнд должен быть целого или вещественного типа, а может иметь также тип указатель.

**Деление (/) и остаток от деления (%):**

Операция деления применима к операндам арифметического типа. Если оба операнда целочисленные, результат операции округляется до целого числа, в противном случае тип результата определяется правилами преобразования. Операция остатка от деления применяется только к целочисленным операндам. Знак результата зависит от реализации.

1. **Операции. Поразрядные операции. Логические операции.**

**Операции языка Си** предназначены для управления данными (более 40). Для их использования необходимо знать: - синтаксис; - приоритеты; - порядок выполнения. Выражения состоят из операндов, операций, скобок и используются для вычисления некоторого значения определенного типа. Каждый операнд может быть, в свою очередь, выражением или одним из его частных случаев.

**Логические операции:**

Операнды могут иметь арифметический тип или быть указателями, причем операнды могут быть различных типов. Если значение первого операнда достаточно, чтобы определить результат операции, то второй операнд не выполняется.

В Си существует три логические операции:

* Логическая операция НЕ (!) или логическое отрицание – истина, если операнд ложь и наоборот
* Логическая операция И (&&) – истина, если истинны оба операнда
* Логическая операция ИЛИ (||) – истина, если один из операндов истина

И, ИЛИ, НЕТ имеют свои приоритеты. Самый высокий приоритет имеет операция НЕТ, т.е. такая операция выполняется в первую очередь. Более низкий приоритет у операции И, и наконец самый малый приоритет у операции ИЛИ.

**Поразрядные операции**:

Операция поразрядного логического И (&) сравнивает каждый бит первого операнда с соответствующим битом второго операнда. Если оба сравниваемых бита единицы, то соответствующий бит результата устанавливается в 1, в противном случае в 0.

Операция поразрядного логического ИЛИ (|) сравнивает каждый бит первого операнда с соответствующим битом второго операнда. Если любой (или оба) из сравниваемых битов равен 1, то соответствующий бит результата устанавливается в 1, в противном случае результирующий бит равен 0.

Операция поразрядного исключающего ИЛИ (^) сравнивает каждый бит первого операнда с соответствующими битами второго операнда. Если один из сравниваемых битов равен 0, а второй бит равен 1, то соответствующий бит результата устанавливается в 1, в противном случае, т.е. когда оба бита равны 1 или 0, бит результата устанавливается в 0.

1. **Операции. Приоритеты операций. Операция определения размера.**

**Операции языка Си** предназначены для управления данными (более 40). Для их использования необходимо знать: - синтаксис; - приоритеты; - порядок выполнения. Выражения состоят из операндов, операций, скобок и используются для вычисления некоторого значения определенного типа. Каждый операнд может быть, в свою очередь, выражением или одним из его частных случаев.

В следующей таблице перечислены приоритеты и ассоциативность операторов в C. Операторы перечислены сверху вниз в порядке уменьшения приоритета.

|  |  |  |  |
| --- | --- | --- | --- |
| **Приоритет** | **Оператор** | **Описание** | **Ассоциативность** |
| **1** | ++ -- | Суффиксные/постфиксные инкремент и декремент | Слева направо |
| () | Вызов функции |
| [] | Обращение к элементу массива |
| . | Обращение к члену структуры или объединения |
| −> | Обращение к члену структуры или объединения через указатель |
| (*type*){*list*} | Составной литерал(C99) |
| **2** | ++ -- | Префиксные инкремент и декремент | Справа налево |
| + − | Унарные плюс и минус |
| ! ~ | Логическое НЕ и побитовое НЕ |
| (*type*) | Приведение типа |
| \* | Разыменование |
| & | Взятие адреса |
| sizeof | Размер |
| \_Alignof | Выравнивание(C11) |
| **3** | \* / % | Умножение, деление и остаток | Слева направо |
| **4** | + − | Сложение и вычитание |
| **5** | << >> | Побитовые левый сдвиг и правый сдвиг |
| **6** | < <= | Для операторов сравнения < и ≤ соответственно |
| > >= | Для операторов сравнения > и ≥ соответственно |
| **7** | == != | Для сравнений = и ≠ соответственно |
| **8** | & | Побитовое И |
| **9** | ^ | Побитовое XOR (исключающее или) |
| **10** | | | Побитовое ИЛИ (включающее или) |
| **11** | && | Логическое И |
| **12** | || | Логическое ИЛИ |
| **13** | ?: | Тернарное условие | Справа налево |
| **14** | = | Простое присваиваиние |
| += −= | Присваивание через сумму и разность |
| \*= /= %= | Присваивание через произведение, частное и остаток |
| <<= >>= | Присваивание через левый сдвиг и правый сдвиг |
| &= ^= |= | Присваивание через побитовые И, исключающее ИЛИ и ИЛИ |
| **15** | , | Запятая | Слева направо |

При лексическом разборе выражения любой оператор из некоторой строки будет теснее привязан (как при использовании скобок) к своим аргументам, чем любой другой, находящийся в строке ниже.

Операторы из одной ячейки (в одной ячейке могут быть перечислены несколько строк операторов) имеют одинаковый приоритет и группируются в указанном направлении. Например, выражение a=b=c разбирается как a=(b=c), а не как (a=b)=c из-за ассоциативности справа налево. Примечание: это не влияет на [порядок вычисления](https://ru.cppreference.com/w/c/language/eval_order" \o "c/language/eval order) подвыражений a, b и c.

**Операция определения размера sizeof :** Она предназначена для вычисления размера объекта или типа в байтах, и имеет две формы:

sizeof выражение

sizeof ( тип )

1. **Операции. Унарные. Бинарные и тернарные операции.**

**Операции языка Си** предназначены для управления данными (более 40). Для их использования необходимо знать: - синтаксис; - приоритеты; - порядок выполнения. Выражения состоят из операндов, операций, скобок и используются для вычисления некоторого значения определенного типа. Каждый операнд может быть, в свою очередь, выражением или одним из его частных случаев.

Операции можно разделить на три группы:

**1) Унарные**

Унарные операции требуют один объект:

<оператор> <операнд> или <операнд> <оператор>

Примеры:

:: доступ к области видимости

. выбор

-> выбор

[ ] индексация

( ) вызов функции

<тип>( ) конструирование

++ инкремент

-- декремент

sizeof размер объекта или типа

~ поразрядное отрицание

! логическое отрицание

- арифметическое отрицание (унарный минус)

+ унарный плюс

& взятие адреса

\* разадресация

new выделение памяти

delete освобождение памяти

(<тип>) преобразование типа

.\* и ->\* выбор

**2) Бинарные**

Бинарные операции требуют двух объектов:

<операнд> <оператор><операнд>

Примеры:

\* умножение

/ деление

% остаток от деления

+ сложение

— вычитание

<< сдвиг влево

>> сдвиг вправо

< меньше

<= меньше или равно

> больше

>= больше или равно

== равно

!= не равно

& поразрядная конъюнкция (И)

^ поразрядное исключающее ИЛИ

| поразрядная дизъюнкция (ИЛИ)

&& логическое И

|| логическое ИЛИ

? : условная операция (тернарная)

= присваивание

\*= умножение с присваиванием

/= деление с присваиванием

%= остаток отделения с присваиванием

+= сложение с присваиванием

-= вычитание с присваиванием

<<= сдвиг влево с присваиванием

>>= сдвиг вправо с присваиванием

&= поразрядное И с присваиванием

|= поразрядное ИЛИ с присваиванием

^= поразрядное исключающее ИЛИ с присваиванием

, последовательное вычисление

**3) Тернарные**

Тернарные операции требуют трех объектов (условная операция). Формат написания условной операции следующий:

**Условие? Оператор 1 : оператор 2;**

если выражение 1 отлично от нуля (Истинно), то результатом операции является выражение 2, в противном случае - результатом операции является выражения 3. Каждый раз вычисляется только одно из выражений 2 или 3.

1. **Основные типы данных. Спецификаторы типов.**

В языке СИ существует несколько типов данных. Каждый тип данных определяется одним из следующих ключевых слов:

1. int (целый) – целые числа. Диапазон возможных целых значений лежит в пределах от -32768 до 32767, переменная типа int занимает 16 бит;

2. char (символьный) – задает значения, которые представляют различные символы;

3. float (вещественный) – задает значения, к которым относятся вещественные числа, имеющие дробную часть, отделяемую точкой. Вещественные числа могут быть записаны также в экспоненциальной форме. Например, -1.58е+2 (что равно -1,58·10 2 ). В языке СИ переменная типа float занимает 32 бита. Она может принимать значения в диапазоне от +3.4е-38 до +3.4е+38;

4. double (двойная точность) – определяет вещественные переменные, занимающие в два раза больше места, чем переменная типа float. Переменная типа double занимает 64 бита. Она может принимать значения в диапазоне от +1.7е-308 до +1.7е+308.

5. void – неопределенный (несуществующий) тип. Объектов типа void не существует. Множество значений этого типа пусто, т.е. нельзя переменной такого типа присвоить какое-нибудь значение. Более того, нельзя даже описать переменную этого типа. С его помощью задаются пустой список аргументов функции или функции, не возвращающие значение.

В свою очередь, данные целого типа могут быть короткими (short), длинными (long) и беззнаковыми (unsigned).

7. short (короткий целый) – соответствующие объекты не могут быть больше, чем int, переменные этого типа занимают 16 бит;

8. long (длинный целый) – соответствующие объекты не могут быть меньше, чем int. Переменная типа long занимает 32 бита и позволяет представить целые числа от –2147483648 до 2147483647;

9. unsigned (беззнаковый) – в языке СИ можно объявлять некоторые типы (char, short, int, long) беззнаковыми с помощью модификатора unsigned (например, unsigned short). Это значит, что соответствующие переменные не будут иметь отрицательных значений. В результате они могут принимать большие положительные значения, чем переменные знаковых типов. В случае типа int объявления вида «unsigned int a;» можно записать «unsigned a;»;

Существует четыре спецификатора **(спецификатор – одно или несколько ключевых слов, определяющих тип переменной)** типа, уточняющих внутреннее представление и диапазон значений стандартных типов: short (короткий); long (длинный); signed (знаковый); unsigned (беззнаковый). Если ключевое слово отсутствует, то класс памяти определяется по контексту.

Данные целого и вещественного типов находятся в определенных числовых диапазонах так как занимают разный объем оперативной памяти

1. **Переменные и выражения.**

**Переменная** – это выделенная нами область в памяти компьютера (выделяем ее с помощью объявления в коде программы), в которой будут храниться наши данные.

Переменные можно изменять. При задании значения переменной в соответствующую ей область памяти помещается код этого значения. Доступ к значению возможен через имя переменной, а доступ к участку памяти - по его адресу.

Каждая переменная перед использованием в программе должна быть определена, т. е. ей должна быть выделена память.

Размер участка памяти, выделяемой для переменной и интерпретация содержимого зависят от типа, указанного в определении переменной.

Простейшая форма определения переменных:

Тип список имен переменных;

Из констант, переменных, разделителей и знаков операций можно конструировать **выражения**. Каждое выражение состоит из одного или нескольких операндов, символов операций и ограничителей, в качестве которых чаще всего выступают квадратные скобки. Если выражение формирует целое или вещественное число, то это арифметическое выражение.

В арифметических выражениях допустимы операции: +, -, \*, /, %.

Отношение – это пара арифметических выражений, объединенных знаком операции отношения. Логический тип в Си отсутствует, поэтому принято, что отношение имеет ненулевое значение, если оно истинно и 0, если оно ложно.

1. **Поразрядные операции. Логические операции. Операции присваивания.**

**Операции языка Си** предназначены для управления данными (более 40). Для их использования необходимо знать: - синтаксис; - приоритеты; - порядок выполнения. Выражения состоят из операндов, операций, скобок и используются для вычисления некоторого значения определенного типа. Каждый операнд может быть, в свою очередь, выражением или одним из его частных случаев.

**Логические операции:** Операнды могут иметь арифметический тип или быть указателями, причем операнды могут быть различных типов. Если значение первого операнда достаточно, чтобы определить результат операции, то второй операнд не выполняется.

В Си существует три логические операции:

* Логическая операция И(&&)-истина, если истинны оба операнда
* Логическая операция ИЛИ(||)-истина, если один из операндов истина;
* Логическая операция НЕ(!) или логическое отрицание – истина, если операнд ложь и наоборот.

**Поразрядныe операции**:

Операция поразрядного логического И (&) сравнивает каждый бит первого операнда с соответствующим битом второго операнда. Если оба сравниваемых бита единицы, то соответствующий бит результата устанавливается в 1, в противном случае в 0.

Операция поразрядного логического ИЛИ (|) сравнивает каждый бит первого операнда с соответствующим битом второго операнда. Если любой (или оба) из сравниваемых битов равен 1, то соответствующий бит результата устанавливается в 1, в противном случае результирующий бит равен 0.

Операция поразрядного исключающего ИЛИ (^) сравнивает каждый бит первого операнда с соответствующими битами второго операнда. Если один из сравниваемых битов равен 0, а второй бит равен 1, то соответствующий бит результата устанавливается в 1, в противном случае, т.е. когда оба бита равны 1 или 0, бит результата устанавливается в 0.

Операции по количеству операндов бывают унарные(один операнд), бинарные(два операнда), тернарная(три операнда).

**Операции присваивания** (=, +=, -=, \*= и т. д.)

Формат операции присваивания: < ID > = <выражение>

Операции присваивания могут использоваться в программе как законченные операторы.

Формат операции простого присваивания (=):

операнд\_1 = операнд\_2

Первый операнд должен быть L-значением, второй — выражением. Сначала вычисляется выражение, стоящее в правой части операции, а потом его результат записывается в область памяти, указанную в левой части (мнемоническое правило: «присваивание — это передача данных «налево»»). То, что ранее хранилось в этой области памяти, естественно, теряется.

В сложных операциях присваивания ( +=, \*=, /= и т п.) при вычислении выражения, стоящего в правой части, используется и L-значение из левой части. Например, при сложении с присваиванием ко второму операнду прибавляется первый, и результат записывается в первый операнд, то есть выражение а += b является более компактной записью выражения а = а + b.

1. **Правила применения символов и выполнения схем. Описание символов (символы данных, символы процесса, символы линий, специальные символы)**

**ПРАВИЛА ПРИМЕНЕНИЯ СИМВОЛОВ И ВЫПОЛНЕНИЯ СХЕМ**

**По ГОСТ 19.701**

1.2. Символы в схеме должны быть расположены равномерно. Следует придерживаться разумной длины соединений и минимального числа длинных линий.

1.3. Не должны изменяться углы и другие параметры, влияющие на соответствующую форму символов. Символы должны быть, по возможности, одного размера.

1.4. Минимальное количество текста, необходимого для понимания функции данного символа, следует помещать внутри данного символа.

Если объем текста, помещаемого внутри символа, превышает его размеры, следует использовать символ комментария.

1.5. В схемах может использоваться идентификатор символов. Это связанный с данным симво­лом идентификатор, который определяет символ для использования в справочных целях в других элементах документации (например, в листинге программы). Идентификатор символа должен распола­гаться слева над символом.

1.6. В схемах может использоваться описание символов. Описание символа должно быть расположено справа над симво­лом.

1.8. В схемах может использоваться подробное представление, которое обозначается с помощью символа с полосой для процесса или данных. Символ с полосой указывает, что в этом же комплекте документации в другом месте имеется более подробное представление.

**2. Правила выполнения соединений**

2.1. Потоки данных или потоки управления в схемах показываются линиями. Направление потока слева направо и сверху вниз считается стандартным.

В случаях, когда необходимо внести большую ясность в схему (например, при соединениях), на линиях используются стрелки.

2.2. В схемах следует избегать пересечения линий.

2.3. Две или более входящие линии могут объединяться в одну исходящую линию. Если две или более линии объединяются в одну линию, место объединения должно быть смещено.

2.4. Линии в схемах должны подходить к символу либо слева, либо сверху, а исходить либо справа, либо снизу. Линии должны быть направлены к центру символа.

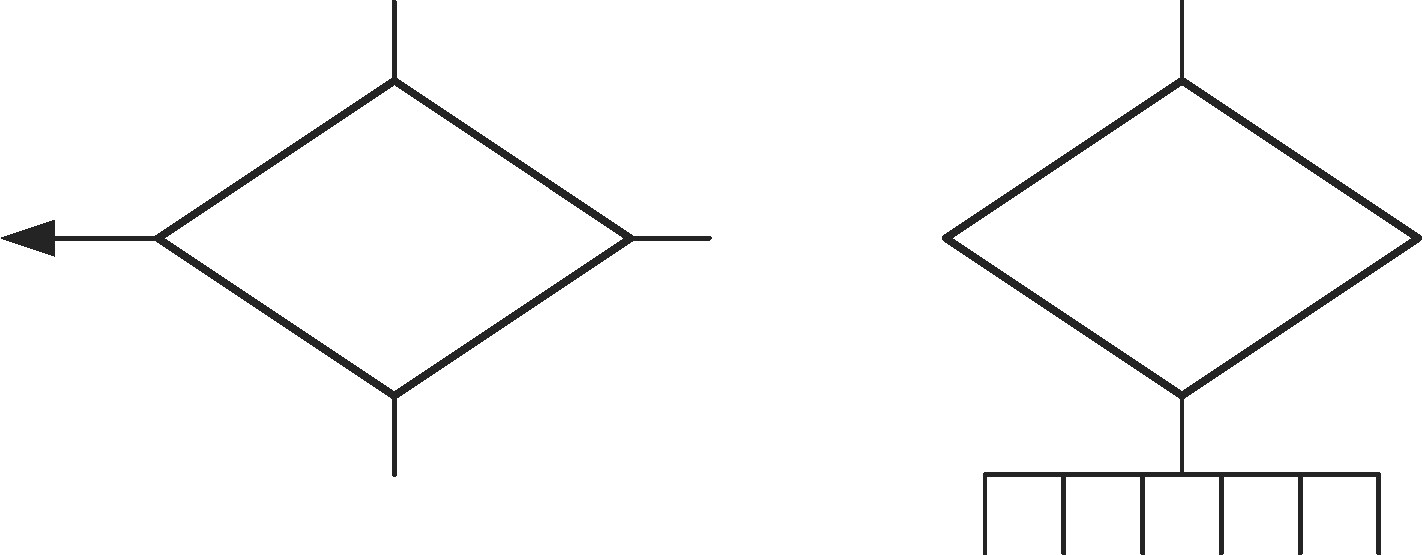
**3. Специальные условные обозначения**

3.1. Несколько выходов

3.1.1. Несколько выходов из символа следует показывать:

1) несколькими линиями отданного символа к другим символам;

2) одной линией от данного символа, которая затем разветвляется в соответствующее число линий.



|  |  |  |
| --- | --- | --- |
| 1. Ввод-вывод (данные) |  | Преобразование данных в форму, пригодную для обработки (ввод) или отображения результатов обработки (вывод) |
| 2. Ручной ввод |  | Ввод данных вручную при помощи неавтономных устройств с клавиатурой, переключателей, кнопок |
| 4. Процесс |  | Выполнение операции или группы операций, в результате которых изменяется значение, форма представления или расположение данных |
| 5. Предопределенный процесс | предопределенный процесс | Использование раннее созданных и отдельно описанных алгоритмов или программ |
| 6. Ручная операция |  | Автономный процесс, выполняемый вручную или при помощи неавтоматически действующих средств |
| 8. Решение |  | Выбор направления выполнения алгоритма или программы в зависимости от некоторых переменных условий |
| 9. Линия потока |  | Поток данных или управления |
| 10. Пунктирная линия |  | Альтернативная связь между двумя или более символами |
| 13. Соединитель |  | Указание связи между прерванными линиями потока, связывающими символы |
| 14. Пуск - терминатор |  | Начало, конец, прерывание процесса обработки данных или выполнения программы |
| 15. Комментарий |  | Связь между элементом схемы и пояснением |
| 16. Граница цикла |  |  |

ВНИМАНИЕ !!! Размер *а* должен выбираться из ряда 10, 15, 20 мм. Допускается увеличивать размер *а* на число, кратное 5. Размер *b=1,5а*.

**29. Правило общих арифметических преобразований. Приоритеты операций.**

Арифметические операции – бинарные. Перечень арифметических операций и их обозначений: + сложение; - вычитание (либо унарная операция - изменение знака); / деление (для int операндов - с отбрасыванием остатка); \* умножение; % остаток от деления целочисленных операндов со знаком первого операнда (деление по модулю).

Арифметические преобразования приводят оба операнда бинарного арифметического выражения к одному типу, который и будет типом результата выражения. Два общих правила таковы:

* типы всегда приводятся к тому из типов, который способен обеспечить наибольший диапазон значений при наибольшей точности. Это помогает уменьшить потери точности при преобразовании;
* любое арифметическое выражение, включающее в себя целые операнды типов, меньших чем int, перед вычислением всегда преобразует их в int.

Рассмотрим общие арифметические преобразования.

1. Операнды типа float преобразуются к типу double.

2. Если один операнд long double, то второй преобразуется к этому же типу.

3. Если один операнд double, то второй также преобразуется к типу double.

4. Любые операнды типа char и short преобразуются к типу int.

5. Любые операнды unsigned char или unsigned short преобразуются к типу unsigned int.

6. Если один операнд типа unsigned long, то второй преобразуется к типу unsigned long.

7. Если один операнд типа long, то второй преобразуется к типу long.

8. Если один операнд типа unsigned int, то второй операнд преобразуется к этому же типу.

Приоритеты арифметических операций:

1.Сложение\вычитание

2. Умножение, деление и остаток ( \*  /  %)

Порядок выполнения операций можно определять круглыми скобками, тогда выражение в скобках выполняется в первую очередь (слева направо)).

**30. Преобразование типов. Преобразование целых типов со знаком. Преобразование целого типа без знака. Преобразование плавающих типов к целым.**

Выделяют **явное и неявное** приведения типов.

При записи выражений желательно использование однородных данных, т.е. переменных и констант имеющих один и тот же тип.

Приведение типа — преобразование значения переменной одного типа в значение другого типа.

**Преобразование целых типов со знаком**. Целое со знаком преобразуется к более короткому целому со знаком, посредством усечения старших битов. Целая со знаком преобразуется к более длинному целому со знаком, путем размножения знака. При преобразовании целого со знаком к целому без знака, целое со знаком преобразуется к размеру целого без знака и результат рассматривается как значение без знака. Преобразование целого со знаком к плавающему типу происходит без потери информации, за исключением случая преобразования значения типа **long int** или unsigned **long int** к типу **float**, когда точность часто может быть потеряна.

**Преобразование целых типов без знака.** Целое без знака преобразуется к более короткому целому без знака или со знаком путем усечения старших битов. Целое без знака преобразуется к более длинному целому без знака или со знаком путем дополнения нулей слева. Когда целое без знака преобразуется к целому со знаком того же размера, битовое представление не изменяется. Поэтому значение, которое оно представляет, изменяется, если знаковый бит установлен (равен 1), т.е. когда исходное целое без знака больше чем максимальное положительное целое со знаком, такой же длины.

**Преобразования плавающих типов**. Величины типа **float** преобразуются к типу **double** без изменения значения. Величины **double** и **long double** преобразуются к **float** c некоторой потерей точности. Если значение слишком велико для float, то происходит потеря значимости, о чем сообщается во время выполнения.

При преобразовании величины с плавающей точкой к целым типам она сначала преобразуется к типу **long** (дробная часть плавающей величины при этом отбрасывается), а затем величина типа **long** преобразуется к требуемому целому типу. Если значение слишком велико для **long**, то результат преобразования не определен.

Преобразования из **float,** **double** или **long double** к типу **unsigned long** производится с потерей точности, если преобразуемое значение больше, чем максимально возможное положительное значение, представленное типом long.

**31. Приведение типов.**

Приведение типа — преобразование значения переменной одного типа в значение другого типа. Выделяют **явное и неявное** приведения типов.

Неявное преобразование типов данных выполняет компилятор С++, ну а явное преобразование данных выполняет сам программист. О преобразовании типов данных скажу следующее: «Результат любого вычисления будет преобразовываться к наиболее точному типу данных, из тех типов данных, которые участвуют в вычислении».

При выполнении операций происходят неявные преобразования типов в следующих случаях:

* при выполнении операций осуществляются обычные арифметические преобразования (которые были рассмотрены выше);
* при выполнении операций присваивания, если значение одного типа присваивается переменной другого типа;
* при передаче аргументов функции.

При явном приведении перед выражением следует указать в круглых скобках имя типа, к которому необходимо преобразовать исходное значение.

При неявном приведении преобразование происходит автоматически, по правилам, заложенным в языке Си.

### ПРИМЕР ЯВНОГО ПРИВЕДЕНИЯ ТИПА

**int x = 5;  
double y = 15.3;  
x = (int) y;**  
**y = (double) x;**

### ПРИМЕР НЕЯВНОГО ПРИВЕДЕНИЯ ТИПА

**int x = 5;  
double y = 15.3;**  
**y = x;** //здесь происходит неявное приведение типа к double  
**x = y;** //здесь происходит неявное приведение типа к int

**32. Принципы работы компьютера**

Компьютер – это техническое средство преобразования информации, в основу работы которого заложены те же принципы обработки электрических сигналов, что и в любом электронном устройстве:

1. входная информация, представленная различными физическими процессами, как электрической, так и неэлектрической природы (буквами, цифрами, звуковыми сигналами и т.д.), преобразуется в электрический сигнал;
2. сигналы обрабатываются в блоке обработки;
3. с помощью преобразователя выходных сигналов обработанные сигналы преобразуются в неэлектрические сигналы (изображения на экране).

Назначение компьютера – обработка различного рода информации и представление ее в удобном для человека виде.

Компьютер работает под управлением некоторой программы, которая предварительно загружается в память с помощью загрузчика, входящего в состав ОС. Программа обращается к содержимому других ячеек памяти и устройствам ввода-вывода для получения данных и сохранения результатов работы программы.

С позиции функционального назначения компьютер – это система, состоящая из 4-х основных устройств, выполняющих определенные функции: запоминающего устройства или памяти, которая разделяется на оперативную и постоянную, арифметико-логического устройства (АЛУ), устройства управления (УУ) и устройства ввода-вывода (УВВ).

Запоминающее устройство (память) предназначается для хранения информации и команд программы в ЭВМ. Информация, которая хранится в памяти, представляет собой закодированные с помощью 0 и 1 числа, символы, слова, команды, адреса и т.д.

**Память** компьютера разбита на ячейки-байты (один байт содержит 8 бит), можно обращаться к ячейке по её адресу – номеру байта в памяти. В компьютерах с 32-битным процессором максимальный объем адресуемой памяти равен 4 Гб, а с 64-битным – 192 Гб (зависит от ОС). Данные в памяти хранятся в нескольких последовательных байтах, например, для хранения целых чисел нужно 4 байта, а вещественных – 8 байт.

Арифметико-логическое устройство (АЛУ). Производит арифметические и логические действия. Следует отметить, что любую арифметическую операцию можно реализовать с использованием операции сложения.

УУ выполняет следующий цикл действий:

1. формирование адреса очередной команды;
2. чтение команды из памяти и ее расшифровка;
3. выполнение команды.

В современных компьютерах функции УУ и АЛУ выполняет одно устройство, называемое центральным процессором.

**Центральный процессор** извлекает очередную команду программы и все необходимые данные для ее исполнения из памяти, выполняет её и записывает при необходимости результаты обратно в память.

Данные для программы могут размещаться в регистровой, статической, стековой и динамической памяти. Отличия между этими видами памяти состоит в способе адресации и времени жизни данных.

**Регистры.**

В центральном процессоре есть небольшое количество регистров (обычно от 4 до 32), которые предназначены для ускорения выполнения команд за счет уменьшения обращений к памяти за данными;

для уменьшения сложности и размеров команд за счет того, что только один из операндов может быть адресом ячейки в памяти.

Форматы машинных команд:

В Си с помощью описателя register можно указать, что некоторую часто используемую переменную следует разместить в регистре. Но данный описатель способа хранения можно считать устаревшим, так как, во-первых, оптимизирующие компиляторы эффективно используют регистры и могут размещать переменную в регистре только на время ее частого использования, а во-вторых, в современных процессорах реализовано многоуровневое кэширование, существенно уменьшающее время обращения к памяти.

**Стек.**

Стек (stack – стопка) – это область памяти, в которую добавление и удаление данных происходит с одной стороны. Регистр SP указывает на вершину стека – первую занятую ячейку. При входе в некоторый блок программы данные добавляются в стек, при выходе из блока – удаляются. Данные, размещаемые в стеке, должны иметь фиксированный размер, так как смещение относительного адреса является константой и вычисляется во время компиляции.

Также стек используется для сохранения значений регистров, например, при вызове подпрограммы адрес команды из регистра PC сохраняется в стеке и в PC записывается адрес вызванной подпрограммы, а возврат из подпрограммы заключается в загрузке вершины стека в регистр PC.

**Куча.**

Куча позволяет во время исполнения программы создавать и уничтожать данные произвольного размера в произвольном порядке. Для повторного использования освобожденной памяти поддерживается список свободных областей памяти, в котором при выделении памяти происходит поиск подходящей по размерам области, а если такой области не будет найдено, запрашивается новая область памяти у операционной системы. Количество действий для этого достаточно велико, поэтому выделение и освобождение памяти из кучи являются медленными операциями.

**33. Приоритет в выражениях. Приоритет операций.**

**Операции языка Си** предназначены для управления данными (более 40). Выражения состоят из операндов, операций, скобок и используются для вычисления некоторого значения определенного типа. Каждый операнд может быть, в свою очередь, выражением или одним из его частных случаев.

Операции, применяемые к одному операнду, - унарные, к двум операндам – бинарные, есть операция с тремя операндами - тернарная.

Операции выполняются в соответствии с приоритетами. Высший приоритет у унарных операций, затем выполняются бинарные, потом логические и тернарные. Для изменения порядка выполнения операций используются круглые скобки.

Большинство операций выполняются слева направо. Исключение:

унарные операции, операции присваивания и тернарная операция ( ? : ) - справа налево.

В следующей таблице перечислены приоритеты и ассоциативность операторов в C. Операторы перечислены сверху вниз в порядке уменьшения приоритета.

|  |  |  |  |
| --- | --- | --- | --- |
| **Приоритет** | **Оператор** | **Описание** | **Ассоциативность** |
| **1** | ++ -- | Суффиксные/постфиксные инкремент и декремент | Слева направо |
| () | Вызов функции |
| [] | Обращение к элементу массива |
| . | Обращение к члену структуры или объединения |
| −> | Обращение к члену структуры или объединения через указатель |
| (*type*){*list*} | Составной литерал(C99) |
| **2** | ++ -- | Префиксные инкремент и декремент | Справа налево |
| + − | Унарные плюс и минус |
| ! ~ | Логическое НЕ и побитовое НЕ |
| (*type*) | Приведение типа |
| \* | Разыменование |
| & | Взятие адреса |
| sizeof | Размер |
| \_Alignof | Выравнивание(C11) |
| **3** | \* / % | Умножение, деление и остаток | Слева направо |
| **4** | + − | Сложение и вычитание |
| **5** | << >> | Побитовые левый сдвиг и правый сдвиг |
| **6** | < <= | Для операторов сравнения < и ≤ соответственно |
| > >= | Для операторов сравнения > и ≥ соответственно |
| **7** | == != | Для сравнений = и ≠ соответственно |
| **8** | & | Побитовое И |
| **9** | ^ | Побитовое XOR (исключающее или) |
| **10** | | | Побитовое ИЛИ (включающее или) |
| **11** | && | Логическое И |
| **12** | || | Логическое ИЛИ |
| **13** | ?: | Тернарное условие | Справа налево |
| **14** | = | Простое присваиваиние |
| += −= | Присваивание через сумму и разность |
| \*= /= %= | Присваивание через произведение, частное и остаток |
| <<= >>= | Присваивание через левый сдвиг и правый сдвиг |
| &= ^= |= | Присваивание через побитовые И, исключающее ИЛИ и ИЛИ |
| **15** | , | Запятая | Слева направо |

При лексическом разборе выражения любой оператор из некоторой строки будет теснее привязан (как при использовании скобок) к своим аргументам, чем любой другой, находящийся в строке ниже.

Операторы из одной ячейки (в одной ячейке могут быть перечислены несколько строк операторов) имеют одинаковый приоритет и группируются в указанном направлении. Например, выражение a=b=c разбирается как a=(b=c), а не как (a=b)=c из-за ассоциативности справа налево. Примечание: это не влияет на [порядок вычисления](https://ru.cppreference.com/w/c/language/eval_order" \o "c/language/eval order) подвыражений a, b и c.

Самый высокий приоритет имеет операция НЕТ, т.е. такая операция выполняется в первую очередь.

Более низкий приоритет у операции И, и наконец самый малый приоритет у операции ИЛИ.

**34. Рекомендации при программировании циклических структур.**

Часто встречающимися *ошибками при программировании циклов* являются использование в теле цикла переменных, которым не были присвоены начальные значения, а также неверная запись условия продолжения цикла.

Чтобы избежать ошибок, рекомендуется:

■ убедиться, что всем переменным, встречающимся в правой части операторов присваивания в теле цикла, до этого присвоены значения, а также возможно ли выполнение других операторов;

■ проверить, изменяется ли в теле цикла хотя бы одна переменная, входящая в условие продолжения цикла;

■ предусматривать аварийный выход из итеративного цикла по достижению некоторого предельно допустимого числа итераций.

■ не следует передавать управление внутрь циклов с помощью оператора goto

Операторы цикла взаимозаменяемы, но можно привести некоторые рекомендации по выбору наилучшего в каждом конкретном случае.

Оператор do while обычно используют, когда цикл требуется обязательно выполнить хотя бы раз (например, если в цикле производится ввод данных).

Оператором while удобнее пользоваться в случаях, когда число итераций заранее не известно, очевидных параметров цикла нет или модификацию параметров удобнее записывать не в конце тела цикла.

Оператор for предпочтительнее в большинстве остальных случаев (однозначно — для организации циклов со счетчиками).

**35. Средства программирования. Этапы получения \*.exe модуля.**

**Средства программирования** — это инструментальные средства, применяемые при разработке программ для микроконтроллеров, микросхем и программирования микросхем памяти. К числу основных инструментальных средств программирования относятся: **универсальные программаторы**, **эмуляторы**, **отладчики**, **внутрисхемные программаторы** и **адаптеры-переходники** для различных корпусов микросхем.

Любая программа на С состоит из файлов(проект). Файлы транслируются компилятором независимо друг от друга, а потом объединяются по строителям задач. В результате создается исполняемый файл (.EXEcutable).

Для того, чтобы выполнить программу требуется перевести её язык на язык, понятный процессору.

1)Включаемые файлы.

2)Исходный код.(\*сpp)

3)Полный текст модуля.

4)Компилятор.

5)Объектный код модуля.(\*.obj)

6)Подключ. библиотеки.(\*.lib)

7)Компоновшик.

8)Исполняемый модуль.(\*.exe)

С помощью текстового редактора разрабатываем программу и сохраняем её в файл.

Сначала программа передаётся препроцессору, который выполняет директивы, содержащиеся в тексте программы. Например: включение в текст программы заголовочных файлов.

В включаемых файлах содержится описание, содержащееся в коде.

Получившийся полный текст программы поступает на вход компилятора, который выделяет лексемы, а затем на основе грамотности языка распознаёт выражение и оператор, построенные из лексем. При этом компиляторы выявляют синтаксические ошибки и в случае их отсутствия строят объектный код. Компоновщик формирует исполняемый модуль файлов, подключая к объектному модулю другие модули, в том числе содержащие функции библиотек, обращение к которым содержится в любой программе. Если программа состоит из несколько исходных файлов, они компилируются по отдельности и объединяются на этапе компоновки. Исполняемый модуль имеет расширение \*.exe.

**36. Ссылки.**

**Ссылка** представляет собой синоним имени, указанного при инициализации ссылки. Ссылку можно рассматривать как указатель, который всегда разыменовывается. Формат объявления ссылки:

тип & имя;

где тип — это тип величины, на которую указывает ссылка, & — оператор ссылки, означающий, что следующее за ним имя является именем переменной ссылочного типа.

**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. Ссылки позволяют использовать в функциях переменные, передаваемые по адресу, без операции разадресации, что улучшает читаемость программы.

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.

int x;

int &y = x;

int z = y;

Если слева от знака присваивания стоит ссылка, то нет никакого способа понять, хотим мы присвоить самой ссылке или объекту, на который она ссылается. Поэтому такое присваивание всегда присваивает объекту, а не ссылке. Но это не относится к инициализации ссылки: инициализируется, разумеется, сама ссылка. Поэтому после инициализации ссылки нет никакого способа изменить её саму, т. е. ссылка всегда постоянна (но не её объект).

**37. Строка. Объявление. Инициализация. Действия над строками и элементами строки.**

Строкой называется массив символов, который заканчивается пустым символом ‘\0’. Строка объявляется как обычный символьный массив, например,

char s1[10]; /\* строка длиной в девять символов \*/

char \*s2; /\* указатель на строку \*/

Различие между указателями s1 и s2 заключается в том, что указатель s1 является именованной константой, а указатель s2 – переменной.

Длина строковой константы не может превышать 509 символов по стандарту. Однако многие реализации допускают строки большей длины.

При инициализации строк размерность массива лучше не указывать, это выполнит компилятор, подсчитав длину строки и добавив к ней единицу. Например,

char s1[ ] = “This is a string.”;

**Действия над строками:**

* 1. **инициализация;**
  2. **ввод/вывод**( gets, puts);
  3. **копирование строк** (strcpy, strncpy);

char \*strcpy(char \*strDst, const char \*strSrc); копирует строку *strSrc* в строку *strDst*

char \*strncpy(char \*strDst, const char \*strSrc, size\_t n);

* 1. **конкатенация или объединение** (strcat, strncat);

char\* strcat(char \*strDst, const char \*strSrc);

присоединяет строку *strSrc* к строке *strDst*, причем завершающий нулевой байт строки *strDst* стирается

* 1. **Преобразование строк в числовые данные**

int atoi (const char \*str);

int n;

char \*str = “-123”;

n = atoi ( str ); /\* n = -123 \*/

* 1. **определение длины строки** (strlen);

int strlen(const char \*str); возвращает длину строки, не учитывая последний нулевой байт.

* 1. **сравнение строк** (strcmp, strncmp);

int strcmp (const char \*str1, const char \*str2);

int strncmp (const char \*str1, const char \*str2, size\_t n);

* 1. **поиск символов в строке** (strchr).

char\* strchr(const char \*str, int c);

ищет первое вхождение символа, заданного параметром *c*, в строку *str*.

* 1. **Поиск подстроки в строке**

char\* strstr(const char \*str1, const char \*str2);

находит первое вхождение строки *str2* (без конечного нулевого байта) в строку *str1*.

**38. Строки. Библиотека string.h. Прототипы функций для работы со строкой.**

**string.h** — [заголовочный файл](https://ru.wikipedia.org/wiki/%2525D0%252597%2525D0%2525B0%2525D0%2525B3%2525D0%2525BE%2525D0%2525BB%2525D0%2525BE%2525D0%2525B2%2525D0%2525BE%2525D1%252587%2525D0%2525BD%2525D1%25258B%2525D0%2525B9_%2525D1%252584%2525D0%2525B0%2525D0%2525B9%2525D0%2525BB) [стандартной библиотеки языка Си](https://ru.wikipedia.org/wiki/%2525D0%2525A1%2525D1%252582%2525D0%2525B0%2525D0%2525BD%2525D0%2525B4%2525D0%2525B0%2525D1%252580%2525D1%252582%2525D0%2525BD%2525D0%2525B0%2525D1%25258F_%2525D0%2525B1%2525D0%2525B8%2525D0%2525B1%2525D0%2525BB%2525D0%2525B8%2525D0%2525BE%2525D1%252582%2525D0%2525B5%2525D0%2525BA%2525D0%2525B0_%2525D1%25258F%2525D0%2525B7%2525D1%25258B%2525D0%2525BA%2525D0%2525B0_%2525D0%2525A1%2525D0%2525B8), содержащий функции для работы с [нуль-терминированными строками](https://ru.wikipedia.org/wiki/%2525D0%25259D%2525D1%252583%2525D0%2525BB%2525D1%25258C-%2525D1%252582%2525D0%2525B5%2525D1%252580%2525D0%2525BC%2525D0%2525B8%2525D0%2525BD%2525D0%2525B8%2525D1%252580%2525D0%2525BE%2525D0%2525B2%2525D0%2525B0%2525D0%2525BD%2525D0%2525BD%2525D0%2525B0%2525D1%25258F_%2525D1%252581%2525D1%252582%2525D1%252580%2525D0%2525BE%2525D0%2525BA%2525D0%2525B0) и различными функциями работы с памятью. Функции, объявленные в string.h, широко используются, так как являясь частью стандартной библиотеки, они гарантированно работают на всех платформах, поддерживающих Си.

1. **Сравнение строк. Для сравнения строк используются функции strcmp и strncmp.**

Функция int strcmp(const char \*str1, const char \*str2); лексикографически сравнивает строки str1 и str2. Функция возвращает 1, 0 или 1, если строка str1 соответственно меньше, равна или больше строки str2.

Функция int strncmp(const char \*str1, const char \*str2, size\_t n); лексикографически сравнивает не более чем n первых символов из строк str1 и str2. Функция возвращает –1, 0 или 1, если первые n символов из строки str1 соответственно меньше, равны или больше первых n символов из строки str2.

2. **Копирование строк. Для копирования строк используются функции strcpy и strncpy.**

Функция char \*strcpy(char \*strDst, const char \*strSrc); копирует строку strSrc в строку strDst. Строка strSrc копируется полностью, включая завершающий нулевой байт. Функция возвращает указатель на strDst. Если строки перекрываются, то результат непредсказуем.

Функция char \*strncpy(char \*strDst, const char \*strSrc, size\_t n); копирует n символов из строки strSrc в строку strDst. Если строка strSrc содержит меньше чем n символов, то последний нулевой байт копируется столько раз, сколько нужно для расширения строки strSrc до n символов. Функция возвращает указатель на строку str1.

4. **Соединение строк. Для соединения строк в одну строку используются функции strcat и strncat.**

Функция char\* strcat(char \*strDst, const char \*strSrc); присоединяет строку strSrc к строке strDst, причем завершающий нулевой байт строки strDst стирается. Функция возвращает указатель на строку strDst.

Функция char\* strncat(char \*strDst, const char \*strSrc, size\_t n); присоединяет n символов из строки strSrc к строке strDst, причем завершающий нулевой байт строки strDst стирается. Если длина строки strSrc меньше n, то присоединяются только символы, входящие в строку strSrc. После соединения строк к строке strDst всегда добавляется нулевой байт. Функция возвращает указатель на строку strDst.

5. **Поиск символа в строке. Для поиска символа в строке используются функции strchr, strrchr, strspn, strcspn и strpbrk.**

Функция char\* strchr(const char \*str, int c); ищет первое вхождение символа, заданного параметром c, в строку str. В случае успеха функция возвращает указатель на первый найденный символ, а в случае неудачи – NULL.

Функция char\* strrchr(const char \*str, int c); ищет последнее вхождение символа, заданного параметром c, в строку str. В случае успеха функция возвращает указатель на последний найденный символ, а в случае неудачи – NULL.

Функция size\_t strspn(const char \*str1, const char \*str2); возвращает индекс первого символа из строки str1, который не входит в строку str2.

Функция size\_t strcspn(const char \*str1, const char \*str2); возвращает индекс первого символа из строки str1, который входит в строку str2.

Функция char\* strpbrk(const char \*str1, const char \*str2); находит в строке str1 первый символ, который равен одному из символов в строке str2. В случае успеха функция возвращает указатель на этот символ, а в случае неудачи – NULL.

**6. Поиск подстроки в строке. Используется функция strstr.**

Функция char\* strstr(const char \*str1, const char \*str2); находит первое вхождение строки str2 (без конечного нулевого байта) в строку str1. В случае успеха функция возвращает указатель на найденную подстроку, а в случае неудачи – NULL. Если указатель str1 указывает на строку нулевой длины, то функция возвращает указатель str1

7. **Разбор строки на лексемы. Для разбора строки на лексемы используется функция strtok.**

Функция char\* strtok(char \*str1, const char \*str2); возвращает указатель на следующую лексему (слово) в строке str1, в которой разделителями лексем являются символы из строки str2. В случае если лексемы закончились, то функция возвращает NULL. При первом вызове функции strtok параметр str1 должен указывать на строку, которая разбирается на лексемы, а при последующих вызовах этот параметр должен быть установлен в NULL. После нахождения лексемы функция strtok записывает после этой лексемы на место разделителя нулевой байт.

**8. Определение длины строки. Для определения длины строки используется функция strlen.**

Функция size\_t strlen(const char \*str); возвращает длину строки, не учитывая последний нулевой байт.

**39. Строки. Функции puts(),fputs(). Функции gets(),fgets().**

Для ввода строки с консоли служит функция

char\* gets(char \*str);

которая записывает строку по адресу str и возвращает адрес введенной строки. Функция прекращает ввод, если встретит символ ‘\n’ или EOF (конец файла). Символ перехода на новую строку не копируется. В конец прочитанной строки помещается нулевой байт. В случае успеха функция возвращает указатель на прочитанную строку, а в случае неудачи NULL.

Для вывода строки на консоль служит стандартная функция

int puts(const char \*s);

которая в случае удачи возвращает неотрицательное число, а в случае неудачи – EOF.

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

Функция fputs() во многом подобна puts(), за тем исключением, что она записывает строку в указанный поток. Функция fgets() читает строку из указанного потока, пока не встретится символ новой строки или не будет прочитано (длина - 1) символов. Если прочитан символ новой строки, то он станет частью строки (в противоположность gets()). В обоих случаях результирующая строка завершается нулевым символом. Функция возвращает str в случае успеха и нулевой указатель - в случае ошибки.  
Можно использовать fgets() как альтернативу gets(). Чтобы сделать это, надо просто указать stdin как указатель на файл.

Преимущество использования fgets() над gets() состоит в том, что можно предотвратить переполнение массива ввода. Массив может содержать символ новой строки.

**40. Структура программы на языке СИ.**

Любая программа на С состоит из файлов(проект). Файлы транслируются компилятором независимо друг от друга, а потом объединяются по строителям задач. В результате создается исполняемый файл (.EXEcutable).

Файлы, содержащие тексты программ на С называются исходными.  
1) заголовочные файлы или h-файлы   
2) файлы реализации(С-файлы)  
Заголовочные имеют расширение «.h” ; файлы реализации имеют расширение «.с» или «.срр»  
Заголовочные файлы содержат только описания(прототипы функций)  
Прототип-описывает имя функции, тип возвращаемого значения, число и типы аргументов. Текста в функции в заголовочном файле НЕТ. Содержатся имена и типы внешних переменных, константы, новые типы, структуры . Заголовочные файлы содержат интерфейсы. Оно сообщают информацию о других программах , при их трансляции никакие объекты не создаются.

Любая программа на С, независимо от размера, состоит из одной или нескольких функций, указывающих фактические операции компьютера, которые должны быть выполнены. Функциям можно давать любые имена, но в программе всегда должна быть функция main, с которой всегда начинается выполнение любой программы. Для выполнения действий main может обращаться к другим функциям, часть из которых находится в той же программе, а часть в библиотеках.

#include <stdio.h> // подключение библиотеки стандартного ввода/вывода

#include <math.h> // подключение математической библиотеки

Int x=1; Int y=2; // определение глобальных переменных

Int sq(int a); // определение прототипа функции

main() // главная функция

{ int b; // описание локальной переменной

b=sq(x+y); // операция(выражение)

printf(‘’b=%d’’,b); } // вывод результата

Int sq(a) { return a\*a } // тело функции sq

Исходная программа состоит из:  
1) директивы, указания компилятором, объявления, опеределения   
2) препроцессор-предварительная обработка текста перед трансляцией

Команды препроцессора называются **директивами**, они начинаются со знака # в начале строки.   
Указания компилятору – команды, выполняемые компилятором во время компиляции.

**Определения** – объявление определяющей переменной и функции. Директивы препроцессора позволяют изменить текст программы, заменить лексемы, вставить текст из другого файла. После директив обязательна ;(точка с запятой)

**Директива #include**   
Включает в текст программы содержимое указанного файла и имеет 2 формы. Имя файла должно соответствовать ОС, может содержать предшествующий маршрут(#include “имя файла”)   
Если в #include использует угловые скобки, то поиск файла будет осуществляться в стандартных директориях среды разработки.

**Директива #define**   
Служит для замены часто используемых констант, ключевых слов, операторов или выражений некоторыми идентификаторами. Идентификаторы могут заменять фрагменты программы и называться макроопределениями, которые могут иметь аргументы.   
#define идентификатор текст   
#define width 80

**Директива #undef**   
Отменяет директиву #define   
#undef идентификатор   
Она отменяет действие текущего определения #define для идентификатора. Не является ошибкой использовать #undef для идентификатора, который не был определён директивой #define

**Комментарии** – текстовые части, предназначенные для аннотирования программы.

Существует 2 способа указания комментария:  
1) многострочный- последовательность символов, помещённые после пары символов /\* , признаком конца является \*/  
Компилятор рассматривает комментарий как символ пробела.   
2) однострочный(//)- начинается с любой позиции строки , и включает в себя все до ее конца .

Язык Си является блочно-структурированным. Каждый блок заключается в фигурные скобки {}.  
Каждое действие в языке Си заканчивается символом "точка с запятой" - ;.

В качестве действия может выступать вызов функции или осуществление некоторых операций.   
Имя функции — это коллективное имя группы описаний и операторов,  
заключенных в блок (фигурные скобки). За именем функции в круглых скобках указываются параметры функции. Если функция не должна возвращать значение, указывается тип void.

**41. Структурное программирование. Основные принципы структурного программирования. Стадии разработки проекта. Критерии качества программ.**

**Структурное программирование** — метод разработки по, в соответствии с которым программа представлена в виде структуры блоков, расположенных в определенном порядке

**Принципы структурного программирования:**

1.Принцип абстракции – программа должна быть спроектирована так, чтобы можно было рассматривать ее с разной степенью детализации.

2.Принцип модульности – программа разделяется на законченные фрагменты (модули), которые можно отлаживать и тестировать независимо друг от друга (отдельные части программы могут реализовываться разными группами людей).

3.Принцип подчиненности – взаимодействие между частями программы должно носить подчиненный характер (к этому приводит принцип проектирования сверху-вниз).

4.Принцип локальности – необходимо стремиться к тому, чтобы каждый модуль использовал свои (локальные) переменные и команды. Желательно не использовать глобальные переменные.

**Стадии разработки проекта:**

1. Постановка задачи.

2. Анализ и исследование задачи, модели.

3. Разработка алгоритма.

4. Программирование.

5. Тестирование и отладка.

6. Анализ результатов решения задачи и уточнение в случае необходимости математической модели с повторным выполнением этапов 2-5.

7. Сопровождение программы (внедрение и поддержка).

**Критерии качества программы:**

Программа является точной, если выдаваемые ею числовые данные имеют допустимые отклонения от аналогичных результатов, полученных с помощью идеальных математических зависимостей.

Программа является совместимой, если она работает должным образом не только автономно, но и как часть программной системы.

Программа является надежной, если она при всех входных данных обеспечивает полную повторяемость результатов.

Программа является универсальной, если она правильно работает при любых допустимых вариантах исходных данных. В ходе разработки программ предусматриваются специальные средства защиты от ввода неправильных данных, обеспечивающие целостность системы.

Программа является защищенной, если она сохраняет работоспособность при возникновении сбоев (режим реального времени, программа большого времени выполнения).

Программа является полезной, если задача, которую она решает, представляет практическую ценность.

Программа является эффективной, если объем требуемых для ее работы ресурсов ЭВМ не превышает допустимого предела.

Программа является проверяемой, если ее качества могут быть продемонстрированы на практике (проверка правильности и универсальности). Существуют формальные математические методы проверки и неформальные (прогоны программы с остановками в контрольных точках, обсуждение результатов заинтересованными пользователями).

Программа является адаптируемой, если она допускает быструю модификацию с целью приспособления к изменяющимся условиям функционирования.

**42. Схема обработки многомерного динамического массива.**

Обработка массивов выполняется обычно следующим образом: объявление, ввод или инициализация элементов массива, преобразование и вывод.

Можно объявлять не только двумерные массивы, но и массивы с большим количеством измерений. Например, объявление int A[n][m][l] создает трехмерный массив из n\*m\*l элементов. Для обращения к каждому элементу такого массива необходимо указать три индекса: A[i][j][k], при этом 0<=i, i<n, 0<=j, j<m, 0<=k, k<l. Количество измерений в массиве может быть практически бесконечным (т.е. достаточным для решения любых практических задач).

Двумерный массив – это массив, в котором каждый элемент является также массивом.

Двумерный массив – это массив, в котором каждый элемент является также массивом. При формировании двумерного динамического массива сначала выделяется память для массива указателей на одномерные массивы, а затем в цикле с параметром выделяется память под одномерные массивы.

Объявление многомерного массива: <тип> < ID >[размер1][размер2]…

Многомерные массивы задаются указанием каждого измерения в квадратных скобках. Например, оператор int matr [6] [8]; задает описание матрицы из 6 строк и 8 столбцов. Нумерация строк и столбцов начинается с 0.

Вообще многомерные массивы, в том числе и двумерные, можно инициализировать как массив массивов. Например, две следующие инициализации равны между собой:

int s[3][5]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}; и

int s[3][5]={{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}};

Во втором случае ввели дополнительные фигурные скобки для записи элементов каждой строки двумерного массива. Если при этом количество элементов в строке не совпадает с числом столбцов в массиве, то соответствующие пропущенные элементы строки считаются неопределенными, так как они не определены.

При формировании двумерного динамического массива сначала выделяется память для массива указателей на одномерные массивы, а затем в цикле с параметром выделяется память под одномерные массивы.

1. int \*\*A = new int\*[N]; // созд-е массива указателей

for (int i = 0; i < N; i++) { //созд-е строк

A[i] = new int[M]; } //N – количество строк; M – количество столбцов

1. int \*\*A = (int \*\*)malloc(N\*sizeof(int \*));

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

A[i] = (int \*)malloc(M\*sizeof(int)); }

Удаление из динамической памяти двумерного массива осуществляется в порядке, обратном его созданию, то есть сначала высвобождается память, выделенная под одномерные массивы с данными, а затем память, выделенная под одномерные массив указателей:

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

delete[] A[i]; } //очищаем память

delete[] A;

1. for (int i=0; i < N; i++)

free(A[i]);

free(A);

Инициализация массива с помощью циклов, элементы генерируются рандомно:

for (int i = 0; i < N; i++) //N – количество строк; M – количество столбцов

{

for (int j = 0; j < M; j++) { //заполняем массив рандомными числами

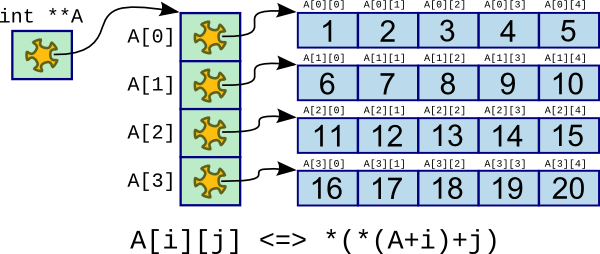
A[i][j] = rand() % 50;

cout << A[i][j] << " "; }

cout << endl;

}

Также можно вводить элементы массива с клавиатуры.

Многомерные массивы в языке Си размещаются в памяти компьютера в последовательности столбцов, поэтому сначала изменяется последний индекс элементов массива. 

**43. Типы с плавающей (фиксированной) точкой. Внутреннее представление.**

В СИ существуют два типа данных с плавающей точкой: float и double. Типы данных с плавающей точкой предназначены для хранения чисел с плавающей точкой. Типы данных float и double могут хранить как положительные, так и отрицательные числа с плавающей точкой. У типа данных float размер занимаемой памяти в два раза меньше, чем у типа данных double, а значит и диапазон принимаемых значений тоже меньше. Если тип данных float объявить с приставкой long, то диапазон принимаемых значений станет равен диапазону принимаемых значений типа данных double. В основном, типы данных с плавающей точкой нужны для решения задач с высокой точностью вычислений, например, операции с деньгами.

Внутреннее представление вещественного числа состоит из 2 частей: мантиссы и порядка. В IBM-совместимых ПК величины типа float занимают 4 байта, из которых один разряд отводится под знак мантиссы, 8 разрядов под порядок и 24 – под мантиссу.

Величины типы double занимают 8 байтов, под порядок и мантиссу отводятся 11 и 52 разряда соответственно. Длина мантиссы определяет точность числа, а длина порядка его диапазон.

**44. Указатели и массивы. Одномерный динамический массив. Выделение/высвобождение памяти.**

При определении массива ему выделяется память. После этого имя массива воспринимается как константный указатель того типа, к которому относятся элементы массива.

Для работы с динамической памятью используют указатели. С их помощью осуществляется доступ к участкам динамической памяти, которые называются динамическими переменными. Динамические переменные создаются с помощью специальных функций и операций. Они существуют либо до конца работы программ, либо до тех пор, пока не будут уничтожены с помощью специальных функций или операций.

Операция new при использовании с массивами возвращает указатель, значением которого служит адрес первого элемента массива. При выделении динамической памяти размеры массива должны быть полностью определены:

int\* a = new int[100];

Имя массива является указателем-константой, значением которой служит адрес первого элемента массива, следовательно, к нему применимы все правила адресной арифметики, связанной с указателями. Запись имя\_массива[индекс] это выражение с двумя операндами: имя массива и индекс. Имя\_массива – это указатель-константа, а индекс определяет смещение от начала массива. Используя указатели, обращение по индексу можно записать следующим образом: \*(имя\_массива+индекс).

for(int i=0;i<n;i++) cout<<\*(a+i)<<" ";

/\*к имени (адресу) массива добавляется константа i и полученное значение разыменовывается\*/

Указатель на динамический массив затем используется при освобождении памяти с помощью операции delete. Примеры: delete[] a;//освобождает память, выделенную под массив, если а адресует его начало

**45. Указатели. Инициализация указателей.**

Указатель — это переменная, содержащая адрес переменной. Размер указателя зависит от модели памяти. Можно определить указатель на указатель: int\*\* a; Указатель может быть константой или переменной, а также указывать на константу или переменную.

int i; //целая переменная

const int ci=1; //целая константа

int\* pi; //указатель на целую переменную

const int\* pci; //указатель на целую константу

int\* const pci; //константный указатель на целую переменную

Указатель можно сразу проинициализировать: int\* pi=&i; //указатель на целую переменную

Для инициализации указателя существуют следующие способы:

1. с помощью операции получения адреса

int a=5;

int \*p=&a; //указателю присвоен адрес а

int p(&a); //указателю присвоен адрес а

2) с помощью проинициализированного указателя int \*r=p;

3) адрес присваивается в явном виде char\*cp=(char\*)0х В800 0000;

где 0х В800 0000 – шестнадцатеричная константа, (char\*) – операция приведения типа.

4) присваивание пустого значения: int\*N=NULL; int \*R=0;

**46. Указатели. Операции над указателями.**

Указатель — это переменная, содержащая адрес переменной.

С указателями можно выполнять следующие операции:

* разыменование (\*);
* присваивание;
* сравнение;
* сложение, вычитание, арифметические инкремент ++, декремент --
* приведение типов.

Операция разыменования предназначена для получения значения переменной или константы, адрес которой хранится в указателе. Если указатель указывает на переменную, то это значение можно изменять, также используя операцию разыменования.

int a; //переменная типа int

int\* pa=new int; //указатель и выделение памяти под динамическую переменную

\*pa=10; //присвоили значение динамической переменной, на которую указывает указатель

a=\*pa; //присвоили значение переменной а

Арифметические операции применимы только к указателям одного типа:

1) Инкремент увеличивает значение указателя на величину sizeof(тип).

2) Декремент уменьшает значение указателя на величину sizeof(тип)

3) Разность двух указателей – это разность их значений, деленная на размер типа в байтах.

4) Суммирование двух указателей не допускается.

5) Можно суммировать указатель и константу.

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).

**47. Указатели. Типы указателей.**

Указатель — это переменная, содержащая адрес переменной.

В языке Си существует три вида указателей:

1. Указатель на объект известного типа. Содержит адрес объекта определенного типа.

2. Указатель типа void. Применяется, еcли тип объекта заранее не определен.

3. Указатель на функцию.

**Указатель на функцию** содержит адрес в сегменте кода, по которому располагается исполняемый код функции, то есть адрес, по которому передается управление при вызове функции. Указатели на функции используются для косвенного вызова функции (не через ее имя, а через обращение к переменной, хранящей ее адрес), а также для передачи имени функции в другую функцию в качестве параметра. Указатель функции имеет тип «указатель функции, возвращающей значение заданного типа и имеющей аргументы заданного типа»:

тип (\*имя) ( список\_типов\_аргументов );

Например, объявление:

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

задает указатель с именем fun на функцию, возвращающую значение типа int и имеющую два аргумента типа double.

**Указатель на объект** содержит адрес области памяти, в которой хранятся данные определенного типа (основного или составного). Простейшее объявление указателя на объект (в дальнейшем называемого просто указателем) имеет вид:

тип \*имя;

где тип может быть любым, кроме ссылки и битового поля, причем тип может быть к этому моменту только объявлен, но еще не определен (следовательно, в структуре, например, может присутствовать указатель на структуру того же типа).

Звездочка относится непосредственно к имени, поэтому для того, чтобы объявить несколько указателей, требуется ставить ее перед именем каждого из них. Например, в операторе

int \*a, b, \*c;

описываются два указателя на целое с именами а и с, а также целая переменная b.

Размер указателя зависит от модели памяти. Можно определить указатель на указатель и т. д.

**Указатель на void** применяется в тех случаях, когда конкретный тип объекта, адрес которого требуется хранить, не определен (например, если в одной и той же переменной в разные моменты времени требуется хранить адреса объектов различных типов). Указателю на void можно присвоить значение указателя любого типа, а также сравнивать его с любыми указателями, но перед выполнением каких-либо действий с областью памяти, на которую он ссылается, требуется преобразовать его к конкретному типу явным образом.

**Указатель на указатель**

В языке Си можно описать переменную типа «указатель на указатель». Это ячейка оперативной памяти, в которой будет храниться адрес указателя на какую-либо переменную. Признак такого типа данных – повторение символа «\*» перед идентификатором переменной.

Указатель на указатель является формой многочисленного перенаправления или цепочки указателей. В случае обычных указателей, указатель содержит адрес некоторого участка памяти, содержащего некоторое значение. В случае указателя на указатель, первый указатель содержит адрес второго, который в свою очередь содержит адрес участка памяти, содержащего некоторое значение.

Например, следующее объявление сообщается компилятору, что amount - это указатель на указатель типа float: float \*\*amount; Важно понимать, что amount - это не указатель на число с плавающей точкой, а указатель на указатель на вещественное число.

**48. Файл math.h, stdlib.h**

#include <stdlib.h> – подключается файл с объявлениями стандартных функций файлового ввода-вывода;

#include <math.h> – объявляет прототипы различных математических функций.

В любой программе кроме операторов и операций используются средства библиотек, входящих в среду программирования, которые облегчают создание программ. Часть библиотек - стандартизована и поставляется с компилятором. В стандартную библиотеку входят функции, макросы, глобальные константы. Это файлы с расширением \*.h, хранящиеся в папке include**.**

**Математические функции** алгоритмического языка Си декларированы в файлах <math.h> и <stdlib.h>. Подключается в начале программы #include <math.h>. Большая часть функции работает с типом double.

**math.h** — заголовочный файл стандартной библиотеки языка программирования С, разработанный для выполнения простых математических операций. Все эти функции принимают double, если не определено иначе. Для работы с типами float и long double используются функции с постфиксами f и l соответственно. Все функции, принимающие или возвращающие угол, работают с радианами. Например (sqrt, fabs, exp, pow, log, tan, sin, acos).

**Математическая запись**

| X |

| X |

arccos X

arcsin X

arctg X

cos X

sin X

tg X

eY

ln X

log X

√X

XY

**Запись на языке СИ**

int abs(int X)

float fabs(float X)

double acos(double X)

double asin(double X)

double atan(double X)

double cos(double X)

double sin(double X)

double tan(double X)

double exp(double X)

double log(double X)

double log10(double X)

double sqrt(double X)

double pow(double X, double Y)

**stdlib.h** — заголовочный файл стандартной библиотеки языка Си, который содержит в себе функции, занимающиеся выделением памяти, контроль процесса выполнения программы, преобразования типов и другие.

Заголовочный файлы stdlib.h определяет макрос NULL, являющийся константным нуль-указателем, который гарантированно указывает на некорректный адрес памяти. stdlib.h определяет тип данных, называемый sizeof.

Единственной исключительной (контролируемой) ситуацией при выполнении арифметических операций является деление на нуль, другие виды ошибочных ситуаций (переполнение, исчезновение порядка или потеря значимости) игнорируются.

**49. Функции форматного ввода/вывода. Спецификаторы. Управляющие символы. Управляющая строка. Спецификаторы формата.**

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

Данные функция определена в библиотеке stdio.h.

Функция **printf()** предназначена для форматированного **вывода**. Она переводит данные в символьное представление и выводит полученные изображения символов на экран. При этом у программиста имеется возможность форматировать данные, то есть влиять на их представление  
на экране.  
  
Общая форма записи функции printf():

printf("управляющая строка", объект1, объект2, ..., объект n);

Управляющая строка состоит из следующих элементов:

* управляющих символов;
* текста, представленного для непосредственного вывода;
* спецификаторы форматов, предназначенных для вывода значений переменных различных типов.

Объекты могут отсутствовать.  
  
**Управляющие символы** не выводятся на экран, а управляют расположением выводимых символов. Отличительной чертой управляющего символа является наличие обратного слэша ‘\’ перед ним.  
Основные управляющие символы:

* ‘\n’ — перевод строки;
* ‘\t’  — горизонтальная табуляция;
* ‘\v’ — вертикальная табуляция;
* ‘\b’ — возврат на символ;
* ‘\r’ — возврат на начало строки;
* ‘\a’ — звуковой сигнал.

**Спецификаторы форматов** нужны для того, чтобы указывать вид, в котором информация будет выведена на экран. Отличительной чертой формата является наличие символа процент ‘%’ перед ним. После знака ‘%’ можно указывать ширину поля под переменную или количество цифр после запятой (через точку): printf(“%6.2d”, S); ширина поля 6 и 2 цифры после запятой.

1. %d – значением аргумента является десятичное целое число;

2. %o – значением аргумента является восьмеричное целое число;

3. %x – значением аргумента является шестнадцатеричное целое число;

4. %c – значением аргумента является символ;

5. %s – значением аргумента является строка символов;

6. %e – значением аргумента является вещественное число в экспоненциальной форме;

7. %f – значением аргумента является вещественное десятичное число с плавающей точкой;

8. %u – значением аргумента является беззнаковое целое число;

9. %p – значением аргумента является указатель (адрес).

## **Ввод информации**

Функция форматированного ввода данных с клавиатуры **scanf()** выполняет чтение данных, вводимых с клавиатуры, преобразует их во внутренний формат и передает вызывающей функции. При этом программист задает правила интерпретации входных данных с помощью спецификаций форматной строки.

Общая форма записи функции scanf( ):

scanf ("Управляющая строка", &объект1, &объект2,...);  
Строка форматов аналогична функции printf().  
Для формирования адреса переменной используется символ амперсанд ‘&’:  
адрес = &объект  
Строка форматов и список аргументов для функции обязательны.

1. **Цели структурного программирования. Принципы структурного программирования. Требования к структурным программам.**

**Структу́рное программи́рование** — метод разработки по, в соответствии с которым программа представлена в виде структуры блоков расположенных в определенном порядке.

**Цели структурного программирования:**

1. повысить надежность программ; для этого нужно, чтобы программа легко поддавалась тестированию и не создавала проблем при отладке. Достигается это хорошим структурированием программы при ее проектировании;
2. повысить эффективность программ; она может быть достигнута при структурировании программы, при разбиении ее на модули так, чтобы можно было бы легко находить и корректировать ошибки, а также чтобы текст любого модуля с целью повышения эффективности его работы можно было переделать независимо от других;
3. уменьшить время и стоимость программной разработки. Достижимо при повышении производительности труда программиста;
4. улучшить читабельность программ; это значит, что необходимо избегать использования языковых конструкций с неочевидной семантикой, стремиться к локализации действия управляющих конструкций и использования структур данных, разрабатывать программу так, чтобы ее можно было бы читать от начала до конца без управляющих переходов на другую страницу;

Разработка программы ведётся пошагово, методом «сверху вниз».

**Принципы структурного программирования:**

1.Принцип абстракции – программа должна быть спроектирована так, чтобы можно было рассматривать ее с разной степенью детализации.

2.Принцип модульности – программа разделяется на законченные фрагменты (модули), которые можно отлаживать и тестировать независимо друг от друга (отдельные части программы могут реализовываться разными группами людей).

3.Принцип подчиненности – взаимодействие между частями программы должно носить подчиненный характер (к этому приводит принцип проектирования сверху-вниз).

4.Принцип локальности – необходимо стремиться к тому, чтобы каждый модуль использовал свои (локальные) переменные и команды. Желательно не использовать глобальные переменные.

**Требования к структурным программам:**

1.Любая программа строится из трёх базовых управляющих конструкций: следрвание, ветвление, цикл.

2.Программа должна разбиваться на независимые модули.

3.Любой модуль должен иметь один вход и один выход.

4.Размер модуля должен быть примерно 30-40 строк.

5.Между модулями устанавливается взаимосвязь по принципу подчиненности.

6.Модуль должен содержать комментарий.

7.Не использовать оператор безусловного перехода (goto).

8.Имена переменных должны нести смысловую нагрузку.

9.В одной строке не следует записывать один оператор.

10.При разработке программ не брать больше 3 уровней (вложений).

11.Избегать использования сложных языковых конструкций.

1. Целый тип. Внутреннее представление.

Целый тип данных предназначен для представления в памяти компьютера обычных целых чисел. Основным и наиболее употребительным целым типом является тип int. Гораздо реже используют его разновидности: short (короткое целое) и long (длинное целое). Также к целым типам относится тип char (символьный). Кроме того, при необходимости можно использовать и тип long long (длинное-предлинное!). По умолчанию все целые типы являются знаковыми (signed), т.е. старший бит в таких числах определяет знак числа: 0 — число положительное, 1 — число отрицательное. Кроме знаковых чисел на Cи можно использовать беззнаковые. В этом случае все разряды участвуют в формировании целого числа. При описании беззнаковых целых переменных добавляется слово unsigned (без знака).

1. int (целый) – целые числа. Диапазон возможных целых значений лежит в пределах от 32768 до 32767, переменная типа int занимает 16 бит;

2. char (символьный) – задает значения, которые представляют различные символы, переменные этого типа занимают 1 байт;

В свою очередь, данные целого типа могут быть короткими (short), длинными (long) и беззнаковыми (unsigned).

3. short (короткий целый) – соответствующие объекты не могут быть больше, чем int, переменные этого типа занимают 16 бит;

4. long (длинный целый) – соответствующие объекты не могут быть меньше, чем int. Переменная типа long занимает 32 бита и позволяет представить целые числа от –2147483648 до 2147483647;

5. unsigned (беззнаковый) – в языке СИ можно объявлять некоторые типы (char, short, int, long) беззнаковыми с помощью модификатора unsigned (например, unsigned short). Это значит, что соответствующие переменные не будут иметь отрицательных значений. В результате они могут принимать большие положительные значения, чем переменные знаковых типов. В случае типа int объявления вида «unsigned int a;» можно записать «unsigned a;».

**Внутреннее представление** величины целого типа — целое число в двоичном коде. При использовании спецификатора signed старший бит числа интерпретируется как знаковый (0 — положительное число, 1 — отрицательное). Спецификатор unsigned позволяет представлять только положительные числа, поскольку старший разряд рассматривается как часть кода числа. Таким образом, диапазон значений типа int зависит от спецификаторов.

1. **Цикл с параметром (For). Инициализации. Модификации. Выражения.**

**Цикл** – организованное повторение некоторой последовательности операторов. Любой цикл обязательно содержит условие прерывания. Операторы, которые повторяются в цикле – тело цикла.

Общий вид оператора: for (инициализация; выражение; модификации) <тело цикла>;

**Инициализация** используется для объявления и присвоения начальных значений величинам, используемым в цикле. В этой части можно записать несколько операторов, разделенных запятой (операцией «последовательное выполнение»). Инициализация выполняется один раз в начале исполнения цикла.

**Выражение** является проверочным условием и выполняется в начале каждой операции цикла. Если оно истинно, то выполняется решение, иначе завершение (условное выражение).

**Модификации** выполняются после каждой итерации цикла и служат обычно для изменения параметров цикла. В части модификаций можно записать несколько операторов через запятую. Простой или составной оператор представляет собой тело цикла.

Любое из выражений в for могут отсутствовать, но ; (точка с запятой) обязательна.

Операция , (запятая) очень часто используется в for. Она позволяет включать в качестве любого из операторов несколько выражений.

for (sum=0 , i=1; i<=N; sum+= i , i++)

Параметры, входящие в выражения, находящиеся в спецификации цикла можно изменять при выполнении операций в коде цикла.

Если отсутствуют выражения 1 и 3, цикл становится эквивалентным while.

Например:

for (;a<20;)

оператор;

1. **Цикл с постусловием**

**Цикл** – организованное повторение некоторой последовательности операторов. Любой цикл обязательно содержит условие прерывания. Операторы, которые повторяются в цикле- тело цикла.

Общий вид записи: **do** *<оператор>* **while** (*<выражение>*);

Цикл do while отличается от [цикла while](http://cppstudio.com/obuchenie_cpp/tsikl-vhile) тем, что в do while сначала выполняется тело цикла, а затем проверяется условие продолжения цикла. Из-за такой особенности do while называют циклом с постусловием.

Тело цикла будет выполняться до тех пор, пока «выражение» истинно. Проверка условия в конце каждой операции. Данный цикл всегда выполняется хотя бы один раз, после чего проверяется, надо ли его выполнять еще раз. Даже если выражение является ложным, то тело цикла выполняется один раз.

**Оператор do while** (цикл с постусловием) обычно используют, когда цикл требуется обязательно выполнить хотя бы раз (например, если в цикле производится ввод данных).

1. **Цикл с предусловием**

**Цикл** – организованное повторение некоторой последовательности операторов. Любой цикл обязательно содержит условие прерывания. Операторы, которые повторяются в цикле- тело цикла.

**Общий вид:** **while** (*<выражение>*) *<оператор>*;

Если выражение принимает истинное значение (не равно 0), то выполняется тело цикла, иначе будет переход к оператору после while. Это повторяется до тех пор, пока выражение не примет значение 0 (ложь). В этом случае выполняется оператор, следующий за while. Проверка выражения осуществляется в начале каждой операции. Если выражение в скобках - ложно (равно 0), то цикл не выполнится ни разу. Среди операторов могут быть **continue** – переход к следующей итерации цикла и **break** – выход из цикла. Для выхода из цикла while при истинности выражения, как и для выхода из других циклов можно пользоваться оператором break.

**Оператором while** (цикл с предусловием)удобнее пользоваться в случаях, когда число итераций заранее не известно, очевидных параметров цикла нет или модификацию параметров удобнее записывать не в конце тела цикла.

1. **Строки. Библиотека string.h. Прототипы функций для работы со строкой.**

**string.h** — [заголовочный файл](https://ru.wikipedia.org/wiki/%2525D0%252597%2525D0%2525B0%2525D0%2525B3%2525D0%2525BE%2525D0%2525BB%2525D0%2525BE%2525D0%2525B2%2525D0%2525BE%2525D1%252587%2525D0%2525BD%2525D1%25258B%2525D0%2525B9_%2525D1%252584%2525D0%2525B0%2525D0%2525B9%2525D0%2525BB) [стандартной библиотеки языка Си](https://ru.wikipedia.org/wiki/%2525D0%2525A1%2525D1%252582%2525D0%2525B0%2525D0%2525BD%2525D0%2525B4%2525D0%2525B0%2525D1%252580%2525D1%252582%2525D0%2525BD%2525D0%2525B0%2525D1%25258F_%2525D0%2525B1%2525D0%2525B8%2525D0%2525B1%2525D0%2525BB%2525D0%2525B8%2525D0%2525BE%2525D1%252582%2525D0%2525B5%2525D0%2525BA%2525D0%2525B0_%2525D1%25258F%2525D0%2525B7%2525D1%25258B%2525D0%2525BA%2525D0%2525B0_%2525D0%2525A1%2525D0%2525B8), содержащий функции для работы с [нуль-терминированными строками](https://ru.wikipedia.org/wiki/%2525D0%25259D%2525D1%252583%2525D0%2525BB%2525D1%25258C-%2525D1%252582%2525D0%2525B5%2525D1%252580%2525D0%2525BC%2525D0%2525B8%2525D0%2525BD%2525D0%2525B8%2525D1%252580%2525D0%2525BE%2525D0%2525B2%2525D0%2525B0%2525D0%2525BD%2525D0%2525BD%2525D0%2525B0%2525D1%25258F_%2525D1%252581%2525D1%252582%2525D1%252580%2525D0%2525BE%2525D0%2525BA%2525D0%2525B0) и различными функциями работы с памятью. Функции, объявленные в string.h, широко используются, так как являясь частью стандартной библиотеки, они гарантированно работают на всех платформах, поддерживающих Си.

1. **Сравнение строк. Для сравнения строк используются функции strcmp и strncmp.**

Функция int strcmp(const char \*str1, const char \*str2); лексикографически сравнивает строки str1 и str2. Функция возвращает 1, 0 или 1, если строка str1 соответственно меньше, равна или больше строки str2.

Функция int strncmp(const char \*str1, const char \*str2, size\_t n); лексикографически сравнивает не более чем n первых символов из строк str1 и str2. Функция возвращает –1, 0 или 1, если первые n символов из строки str1 соответственно меньше, равны или больше первых n символов из строки str2.

**2. Копирование строк. Для копирования строк используются функции strcpy и strncpy.**

Функция char \*strcpy(char \*strDst, const char \*strSrc); копирует строку strSrc в строку strDst. Строка strSrc копируется полностью, включая завершающий нулевой байт. Функция возвращает указатель на strDst. Если строки перекрываются, то результат непредсказуем.

Функция char \*strncpy(char \*strDst, const char \*strSrc, size\_t n); копирует n символов из строки strSrc в строку strDst. Если строка strSrc содержит меньше чем n символов, то последний нулевой байт копируется столько раз, сколько нужно для расширения строки strSrc до n символов. Функция возвращает указатель на строку str1.

**4. Соединение строк. Для соединения строк в одну строку используются функции strcat и strncat.**

Функция char\* strcat(char \*strDst, const char \*strSrc); присоединяет строку strSrc к строке strDst, причем завершающий нулевой байт строки strDst стирается. Функция возвращает указатель на строку strDst.

Функция char\* strncat(char \*strDst, const char \*strSrc, size\_t n); присоединяет n символов из строки strSrc к строке strDst, причем завершающий нулевой байт строки strDst стирается. Если длина строки strSrc меньше n, то присоединяются только символы, входящие в строку strSrc. После соединения строк к строке strDst всегда добавляется нулевой байт. Функция возвращает указатель на строку strDst.

**5. Поиск символа в строке. Для поиска символа в строке используются функции strchr, strrchr, strspn, strcspn и strpbrk.**

Функция char\* strchr(const char \*str, int c); ищет первое вхождение символа, заданного параметром c, в строку str. В случае успеха функция возвращает указатель на первый найденный символ, а в случае неудачи – NULL.

Функция char\* strrchr(const char \*str, int c); ищет последнее вхождение символа, заданного параметром c, в строку str. В случае успеха функция возвращает указатель на последний найденный символ, а в случае неудачи – NULL.

Функция size\_t strspn(const char \*str1, const char \*str2); возвращает индекс первого символа из строки str1, который не входит в строку str2.

Функция size\_t strcspn(const char \*str1, const char \*str2); возвращает индекс первого символа из строки str1, который входит в строку str2.

Функция char\* strpbrk(const char \*str1, const char \*str2); находит в строке str1 первый символ, который равен одному из символов в строке str2. В случае успеха функция возвращает указатель на этот символ, а в случае неудачи – NULL.

**6. Поиск подстроки в строке. Используется функция strstr.**

Функция char\* strstr(const char \*str1, const char \*str2); находит первое вхождение строки str2 (без конечного нулевого байта) в строку str1. В случае успеха функция возвращает указатель на найденную подстроку, а в случае неудачи – NULL. Если указатель str1 указывает на строку нулевой длины, то функция возвращает указатель str1

7. **Разбор строки на лексемы. Для разбора строки на лексемы используется функция strtok.**

Функция char\* strtok(char \*str1, const char \*str2); возвращает указатель на следующую лексему (слово) в строке str1, в которой разделителями лексем являются символы из строки str2. В случае если лексемы закончились, то функция возвращает NULL. При первом вызове функции strtok параметр str1 должен указывать на строку, которая разбирается на лексемы, а при последующих вызовах этот параметр должен быть установлен в NULL. После нахождения лексемы функция strtok записывает после этой лексемы на место разделителя нулевой байт.

**8. Определение длины строки. Для определения длины строки используется функция strlen.**

Функция size\_t strlen(const char \*str); возвращает длину строки, не учитывая последний нулевой байт.

1. **Этапы решения задач на ЭВМ**

**1. Постановка задачи:**

• сбор информации о задаче;

• формулировка условия задачи;

• определение конечных целей решения задачи;

• определение формы выдачи результатов;

• описание данных (их типов, диапазонов величин, структуры и т. п.).

**2. Анализ и исследование задачи, модели:**

• анализ существующих аналогов;

• анализ технических и программных средств;

• разработка математической модели;

• разработка структур данных.

**3. Разработка алгоритма:**

• выбор метода проектирования алгоритма;

• выбор формы записи алгоритма (блок-схемы, псевдокод и др.);

• выбор тестов и метода тестирования;

• проектирование алгоритма.

**4. Программирование:**

• выбор языка программирования;

• уточнение способов организации данных;

• запись алгоритма на выбранном языке программирования.

**5. Тестирование и отладка:**

• синтаксическая отладка;

• отладка семантики и логической структуры;

• тестовые расчеты и анализ результатов тестирования;

• совершенствование программы.

**6. Анализ результатов решения задачи и уточнение в случае необходимости математической модели с повторным выполнением этапов 2-5.**

**7. Сопровождение программы:**

• доработка программы для решения конкретных задач;

• составление документации к решенной задаче, к математической модели, к алгоритму, к программе, к набору тестов, к использованию

58. Оператор cin и cout. Манипуляторы форматирования

Ссылки: https://prog-cpp.ru/cpp-std/

Поточный ввод-вывод в C++ выполняется с помощью функций сторонних библиотек. В С++ разработана библиотека ввода-вывода iostream, использующая концепцию объектно-ориентированного программирования:

#include <iostream>

Библиотека iostream определяет три стандартных потока:

cin  стандартный входной поток (stdin в С)

cout  стандартный выходной поток (stdout в С)

cerr  стандартный поток вывода сообщений об ошибках (stderr в С)

Для выполнения операций ввода-вывода переопределены две операции поразрядного сдвига:

>>  получить из входного потока

<<  поместить в выходной поток

Вывод информации: cout << значение.

Возможно многократное назначение потоков:

cout << 'значение1' << 'значение2' << ... << 'значение n';

Ввод информации: cin >> значение (или cin >> знач1 >> знач2;)

Особого внимания заслуживает ввод символьных строк. По умолчанию потоковый ввод cin вводит строку до пробела, символа табуляции или перевода строки.

Для ввода текста до символа перевода строки используется манипулятор потока getline()

Манипуляторы:

Функцию - манипулятор потока можно включать в операции помещения в поток и извлечения из потока (<<, >>).

В С++ имеется ряд манипуляторов. Рассмотрим основные:

endl – помещение в выходной поток символа конца строки ‘\n’

dec, oct, hex – установка систем счисления.

fill(‘символ’) – заполняет пустые знакоместа значением символа

get() - ожидает ввода символа

getline(указатель, количество) – ожидает ввода строки символов. Максимальное количество символов ограничено полем количество.

59. Оператов cin и cout. Форматирующие функции-члены

Соус: соус

Поточный ввод-вывод в C++ выполняется с помощью функций сторонних библиотек. В С++ разработана библиотека ввода-вывода iostream, использующая концепцию объектно-ориентированного программирования:

#include <iostream>

Библиотека iostream определяет три стандартных потока:

cin  стандартный входной поток (stdin в С)

cout  стандартный выходной поток (stdout в С)

cerr  стандартный поток вывода сообщений об ошибках (stderr в С)

Для выполнения операций ввода-вывода переопределены две операции поразрядного сдвига:

>>  получить из входного потока

<<  поместить в выходной поток

Вывод информации: cout << значение.

Возможно многократное назначение потоков:

cout << 'значение1' << 'значение2' << ... << 'значение n';

Ввод информации: cin >> значение (или cin >> знач1 >> знач2;)

Особого внимания заслуживает ввод символьных строк. По умолчанию потоковый ввод cin вводит строку до пробела, символа табуляции или перевода строки.

Для ввода текста до символа перевода строки используется манипулятор потока getline()

Форматирующие функции-члены:

Доступ к функциям осуществляется через операцию точка, а в круглых скобках передаётся аргумент.

Все их три : width ( ), precision ( ) и fill ( ).

Функция width ( ) позволяет задать минимальную ширину поля для вывода значения. При вводе она задает максимальное число читаемых символов. Если выводимое значение имеет меньше символов, чем заданная ширина поля, то оно дополняется символами-заполнителями до заданной ширины (по умолчанию – пробелами). Однако если выводимое значение имеет больше символов, чем ширина отведенного ему поля, то поле будет расширено до нужного размера.

Функция precision ( ) позволяет узнать или задать точность (число выводимых цифр после запятой), с которой выводятся числа с плавающей точкой. По умолчанию числа с плавающей точкой выводятся с точностью, равной шести цифрам.

Функция fill ( ) позволяет прочесть или установить символ-заполнитель. Например cout.fill(“.”) заполнит все свободные поля(знакоместа) точками.

60. Операторы cout и cin. Флаги и манипуляторы

Соус: http://cppstudio.com/post/319/

Поточный ввод-вывод в C++ выполняется с помощью функций сторонних библиотек. В С++ разработана библиотека ввода-вывода iostream, использующая концепцию объектно-ориентированного программирования:

#include <iostream>

Библиотека iostream определяет три стандартных потока:

cin  стандартный входной поток (stdin в С)

cout  стандартный выходной поток (stdout в С)

cerr  стандартный поток вывода сообщений об ошибках (stderr в С)

Для выполнения операций ввода-вывода переопределены две операции поразрядного сдвига:

>>  получить из входного потока

<<  поместить в выходной поток

Вывод информации: cout << значение.

Возможно многократное назначение потоков:

cout << 'значение1' << 'значение2' << ... << 'значение n';

Ввод информации: cin >> значение (или cin >> знач1 >> знач2;)

Особого внимания заслуживает ввод символьных строк. По умолчанию потоковый ввод cin вводит строку до пробела, символа табуляции или перевода строки.

Для ввода текста до символа перевода строки используется манипулятор потока getline()

Флаги и манипуляторы:

Возможность управлять вводом-выводом в С++, обеспечивают форматирующие функции-члены, флаги и манипуляторы. Флаги, функции и манипуляторы выполняют одну и туже задачу — задают определённый формат ввода/вывода информации в потоках. Ввод/вывод на экран/с экрана в С++ осуществляется с помощью операторов cin и cout соответственно, а значит  манипуляторы форматирования используются совместно с данными операторами ввода/вывода.

Флаги форматирования позволяют включить или выключить один из параметров ввода/вывода.  Чтобы установить флаг ввода/вывода, необходимо вызвать функцию setf(), если необходимо отключить флаг вывода, то используется функция  unsetf().

Пример:

cout.self(ios::/\*имя флага\*/);

Если   при   вводе/выводе   необходимо   установить(снять) несколько   флагов,   то   можно воспользоваться  поразрядной логической операцией  ИЛИ  |.   В  этом   случае   конструкция языка C++ будет такой:

// установка нескольких флагов

cout.setf( ios::/\*name\_flag1\*/ | ios::/\*name\_flag2\*/ | ios::/\*name\_flag\_n\*/ );

Основные флаги:

boolalpha: вывод логических величин в текстовом виде(true, false)

oct, dec, hex – вывод величин в соответствующей системе счисления. (по умолчанию стоит dec. Для изменения сначала надо снять флаг dec, а затем установить oct или hex)

showbase – выводить индикатор основания системы счисления (вывод: 01438 или 9910)

right, left – выравнивание по правой/левой границе. Сначало необходимо установить ширину поля(ширина поля должна быть заведомо больше чем, длина выводимой строки)

Ещё один способ форматирования — форматирование с помощью манипуляторов. Манипулятор — объект особого типа, который управляет потоками ввода/вывода, для форматирования передаваемой в потоки информации. Отчасти манипуляторы дополняют функционал, для форматирования ввода/вывода. Но большинство манипуляторов выполняют точно, то же самое, что и функции с флагами форматирования. Есть случаи, когда проще пользоваться флагами или функциями форматирования, а иногда удобнее использовать манипуляторы форматирования. Именно по этому в С++ предусмотрено несколько средств форматирования ввода/вывода.

Cout << fixed << setprecison(n) << (13/2) << endl;

Итог:

cout.width(); // функция

cout.setf(ios::boolalpha) //флаг

cout << endl; // манипулятор

61. Библиотека <iostream>

Ле соус: https://ru.wikipedia.org/wiki/Iostream

iostream — заголовочный файл с классами, функциями и переменными для организации ввода-вывода в языке программирования C++. Он включён в стандартную библиотеку C++. Название образовано от Input/Output Stream («поток ввода-вывода»). В языке C++ и его предшественнике, языке программирования Си, нет встроенной поддержки ввода-вывода, вместо этого используется библиотека функций. iostream управляет вводом-выводом, как и stdio.h в Си. iostream использует объекты cin, cout, cerr и clog для передачи информации и из стандартных потоков ввода, вывода, ошибок без буферизации и ошибок с буферизацией соответственно. Являясь частью стандартной библиотеки C++, эти объекты также являются частью стандартного пространства имён — std.

Hello world при помощи iostream:

#include <iostream> // в Си: #include <stdio.h>

using namespace std;

int main()

{

cout << "Hello, world!" << endl; // в Си: printf("Hello, World!\n");

return 0;

}

Форматирование вывода осуществляется засчет функций, манипуляторов и флагов.

62. Пространство имен (using namespace std;)

Пространство имен — это декларативная область, в рамках которой определяются различные идентификаторы (имена типов, функций, переменных, и т. д.). Пространства имен используются для организации кода в виде логических групп и с целью избежания конфликтов имен, которые могут возникнуть, особенно в таких случаях, когда база кода включает несколько библиотек. Все идентификаторы в пределах пространства имен доступны друг другу без уточнения. Идентификаторы за пределами пространства имен могут обращаться к членам с помощью полного имени для каждого идентификатора, например std::vector<std::string> vec; , или else с помощью объявления using для одного идентификатора ( using std::string ) или директивы using для всех идентификаторов в пространстве имен ( using namespace std; ). Код в файлах заголовков всегда должен содержать полное имя в пространстве имен.

63. Динамическое выделение памяти для одномерного массива C++

Соус: https://prog-cpp.ru/c-alloc/

Очень часто возникают задачи обработки массивов данных, размерность которых заранее неизвестна. В этом случае возможно использование одного из двух подходов:

выделение памяти под статический массив, содержащий максимально возможное число элементов, однако в этом случае память расходуется не рационально;

динамическое выделение памяти для хранение массива данных.

Динамическое выделение памяти для одномерных массивов:

В Си работать с динамической памятью можно при помощи соответствующих функций распределения памяти (calloc, malloc, free), для чего необходимо подключить библиотекуmalloc.hС++ использует новые методы работы с динамической памятью при помощи операторов new и delete:

new — для выделения памяти;

delete — для освобождения памяти.

Оператор new используется в следующих формах:

new тип; — для переменных

new тип[размер]; — для массивов

Память может быть распределена для одного объекта или для массива любого типа, в том числе типа, определенного пользователем. Результатом выполнения операции new будет указатель на отведенную память, или исключение std::bad\_alloc в случае ошибки.

Память, отведенная в результате выполнения new, будет считаться распределенной до тех пор, пока не будет выполнена операция delete.

delete указатель; — для одного элемента

delete[] указатель; — для массивов

Освобождаться с помощью delete может только память, выделенная оператором new.

Пример:

#include <iostream>using namespace std;int main(){  int size;  int \*dan;  cout << "Ввести размерность массива : ";  cin >> size;  dan = new int[size];  for (int i = 0; i<size; i++) {    cout << "dan[" << i << "] = ";    cin >> dan[i];  }  for (int i = 0; i<size; i++)    cout << dan[i] << " ";  delete[] dan;  return 0;}

64. Динамическое выделение памяти для двумерного массива C++

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.

Выделение памяти под двумерный динамический массив

При формировании двумерного динамического массива сначала выделяется память для

массива указателей на одномерные массивы, а затем в цикле с параметром выделяется

память под одномерные массивы. На рисунке представлена схема динамической области

памяти, выделенной под двумерный массив.

Синтаксис выделения памяти под массив указателей:

ИмяМассива = new Тип \* [ВыражениеТипаКонстанты];

Синтаксис выделения памяти для массива значений:

ИмяМассива[ЗначениеИндекса] = new Тип [ВыражениеТипа

Константы];

ИмяМассива – идентификатор массива, то есть имя двойного указателя для выделяемого

блока памяти.

Тип – тип указателя на массив.

ВыражениеТипаКонстанты – задает количество элементов (размерность) массива.

Выражение константного типа вычисляется на этапе компиляции.

Например:

int n, m;//n и m – количество строк и столбцов матрицы

float \*\*matr; //указатель для массива указателей

matr = new float \* [n]; //выделение динамической памяти

под массив указателей

for (int i=0; i&lt;n; i++)

matr[i] = new float [m]; //выделение динамической памяти

для массива значений

При выделении динамической памяти размеры массивов должны быть полностью

определены.

65. Динамическое выделение памяти для диномической строки C++

Динамический массив(в т.ч. массив char) – это массив, размер которого заранее не фиксирован и может

меняться во время исполнения программы. Для изменения размера динамического

массива язык программирования С++, поддерживающий такие массивы, предоставляет

специальные встроенные функции или операции. Динамические массивы дают

возможность более гибкой работы с данными, так как позволяют не прогнозировать

хранимые объемы данных, а регулировать размер массива в соответствии с реально

необходимыми объемами.

68. Высвобождение динамической памяти, выделенной под одномерный массив

Освобождение памяти, выделенной под одномерный динамический массив,

осуществляется 2 способами.

1) при помощи операции delete, которая освобождает участок памяти ранее выделенной

операцией new.

Синтаксис:

delete [] ИмяМассива;

ИмяМассива – идентификатор массива, то есть имя указателя для выделяемого блока

памяти.

Например:

delete [] mas; /\*освобождает память, выделенную под

массив, если mas адресует его начало\*/

delete [] m;

delete [] lm;

Квадратные скобки [] сообщают оператору, что требуется освободить память, занятую

всеми элементами, а не только первым.

2) при помощи библиотечной функции free, которая служит для освобождения

динамической памяти.

Синтаксис:

free (ИмяМассива);

ИмяМассива – идентификатор массива, то есть имя указателя для выделяемого блока

памяти.

Например:

free (a); //освобождение динамической памяти

69. Высвобождение динамической памяти, выделенной под одномерный массив

Удаление из динамической памяти двумерного массива осуществляется в порядке,

обратном его созданию, то есть сначала освобождается память, выделенная под

одномерные массивы с данными, а затем память, выделенная под одномерные массив

указателей.

Освобождение памяти, выделенной под двумерный динамический массив, также

осуществляется 2 способами.

1) при помощи операции delete, которая освобождает участок памяти ранее выделенной

операцией new.

Синтаксис освобождения памяти, выделенной для массива значений:

delete ИмяМассива [ЗначениеИндекса];

Синтаксис освобождения памяти, выделенной под массив указателей:

delete [] ИмяМассива;

ИмяМассива – идентификатор массива, то есть имя двойного указателя для выделяемого

блока памяти.

Например:

for (int i=0; i&lt;n; i++)

delete matr [i];

//освобождает память, выделенную для массива значений

delete [] matr;

//освобождает память, выделенную под массив указателей

Квадратные скобки [] означают, что освобождается память, занятая всеми элементами

массива, а не только первым.

2) при помощи библиотечной функции free, которая предназначена для освобождения

динамической памяти.

Синтаксис освобождения памяти, выделенной для массива значений:

free (ИмяМассива[ЗначениеИндекса]);

Синтаксис освобождения памяти, выделенной под массив указателей:

free (ИмяМассива);

ИмяМассива – идентификатор массива, то есть имя двойного указателя для выделяемого

блока памяти.

Например:

for (int i=0; i&lt;n; i++)

free (matr[i]);

//освобождает память, выделенную для массива значений

free (matr);

//освобождает память, выделенную под массив указателей

74 75. Заполнение массива случайными значениями

Для написания кода генерации массива случайными целыми числами используется:

1. Функция srand(). Синтаксис:

void srand(unsigned seed);

– функция устанавливает свой аргумент как основу ( seed ) для новой

последовательности псевдослучайных целых чисел, возвращаемых функцией rand().

Сформированную последовательность можно воспроизвести. Для этого необходимо

вызвать srand() с соответствующей величиной seed.

Для использования данной функции необходимо подключить библиотечный файл

<lt;stdlib.h>;.

2. Функция rand(). Синтаксис:

int rand(void);

– функция возвращает псевдослучайное число в диапазоне от нуля до RAND\_MAX. Для

использования данной функции необходимо подключить библиотечный файл

<stdlib.h>;.

3. Константа RAND\_MAX определяет максимальное значение случайного числа, которое

может быть возвращено функцией rand(). Значение RAND\_MAX – это максимальное

положительное целое число.

4. Часто в задачах требуется выполнить генерацию массива на произвольном

промежутке [a,b). Для этого используются следующие выражения:

//генерация случайных целых чисел на [a,b)

x[i]=rand()%(b-a)+a;

//генерация случайных вещественных чисел на [a,b)

y[i]= rand()\*1.0/(RAND\_MAX)\*(b-a)+a;

Пример:

#include <iostream>

using namespace std;

int main()

{

int a[10];

int i;

srand(time(0)); // инициализация генерации случайных чисел

//генерируем целый случайный массив из 10 эелментов от 1 до 5

for (i = 0; i < 10; i++)

a[i] = 1 + rand() % 5;

for (i = 0; i < 10; i++)

cout<<a[i]<<" "; // вывод результата на экране

return 0;

}

76 77. Работа то строками C++. Библиотека. Функции

Строки в языке C++ (класс string)

Основы работы со строками в C++

В языке C++ для удобной работы со строками есть класс string, для использования которого необходимо подключить заголовочный файл string.

Строки можно объявлять и одновременно присваивать им значения:

string S1, S2 = "Hello";

Строка S1 будет пустой, строка S2 будет состоять из 5 символов.

К отдельным символам строки можно обращаться по индексу, как к элементам массива или C-строк. Например S[0] - это первый символ строки.

Для того, чтобы узнать длину строки можно использовать метод size() строки. Например, последний символ строки S это S[S.size() - 1].

Строки в языке C++ могут

Конструкторы строк

Строки можно создавать с использованием следующих конструкторов:string() - конструктор по умолчанию (без параметров) создает пустую строку.string(string & S) - копия строки Sstring(size\_t n, char c) - повторение символа c заданное число n раз.string(size\_t c) - строка из одного символа c.string(string & S, size\_t start, size\_t len) - строка, содержащая не более, чем len символов данной строки S, начиная с символа номер start.

Конструкторы можно вызывать явно, например, так:

S += string(10, 'z');

В этом примере явно вызывается конструктор string для создания строки, состоящей из 10 символов 'z'.

Неявно конструктор вызывается при объявлении строки с указанием дополнительных параметров. Например, так:

string S(10, 'z');

Подробней о конструкторах для строк читайте здесь.

Ввод-вывод строк

Строка выводится точно так же, как и числовые значения:

cout << S;

Для считывания строки можно использовать операцию ">>" для объекта cin:

cin >> S;

В этом случае считывается строка из непробельных символов, пропуская пробелы и концы строк. Это удобно для того, чтобы разбивать текст на слова, или чтобы читать данные до конца файла при помощи while (cin >> S).

Можно считывать строки до появления символа конца строки при помощи функции getline. Сам символ конца строки считывается из входного потока, но к строке не добавляется:

getline(cin S);

Арифметические операторы

Со строками можно выполнять следующие арифметические операции:= - присваивание значения.+= - добавление в конец строки другой строки или символа.+ - конкатенация двух строк, конкатенация строки и символа.==, != - посимвольное сравнение.<, >, <=, >= - лексикографическое сравнение.

То есть можно скопировать содержимое одной строки в другую при помощи операции S1 = S2, сравнить две строки на равенство при помощи S1 == S2, сравнить строки в лексикографическом порядке при помощи S1 < S2, или сделать сложение (конкатенацию) двух строк в виде S = S1 + S2.

Подробней об операторах для строк читайте здесь.

Методы строк

У строк есть разные методы, многие из них можно использовать несколькими разными способами (с разным набором параметров).

Рассмотрим эти методы подробней.

size

Метод size() возращает длину длину строки. Возвращаемое значение является беззнаковым типом (как и во всех случаях, когда функция возращает значение, равное длине строке или индексу элемента - эти значения беззнаковые). Поэтому нужно аккуратно выполнять операцию вычитания из значения, которое возвращает size(). Например, ошибочным будет запись цикла, перебирающего все символы строки, кроме последнего, в виде for (int i = 0; i < S.size() - 1; ++i).

Кроме того, у строк есть метод length(), который также возвращает длину строки.

Подробней о методе size.

resize

S.resize(n) - Изменяет длину строки, новая длина строки становится равна n. При этом строка может как уменьшится, так и увеличиться. Если вызвать в виде S.resize(n, c), где c - символ, то при увеличении длины строки добавляемые символы будут равны c.

Подробней о методе resize.

clear

S.clear() - очищает строчку, строка становится пустой.

Подробней о методе clear.

empty

S.empty() - возвращает true, если строка пуста, false - если непуста.

Подробней о методе empty.

push\_back

S.push\_back(c) - добавляет в конец строки символ c, вызывается с одним параметром типа char.

Подробней о методе push\_back.

append

Добавляет в конец строки несколько символов, другую строку или фрагмент другой строки. Имеет много способов вызова.

S.append(n, c) - добавляет в конец строки n одинаковых символов, равных с. n имеет целочисленный тип, c - char.

S.append(T) - добавляет в конец строки S содержимое строки T. T может быть объектом класса string или C-строкой.

S.append(T, pos, count) - добавляет в конец строки S символы строки T начиная с символа с индексом pos количеством count.

Подробней о методе append.

erase

S.erase(pos) - удаляет из строки S  с символа с индексом pos и до конца строки.

S.erase(pos, count) - удаляет из строки S  с символа с индексом pos количеством count или до конца строки, если pos + count > S.size().

Подробней о методе erase.

insert

Вставляет в середину строки несколько символов, другую строку или фрагмент другой строки. Способы вызова аналогичны способам вызова метода append, только первым параметром является значение i - позиция, в которую вставляются символы. Первый вставленный символ будет иметь индекс i, а все символы, которые ранее имели индекс i и более сдвигаются вправо.

S.insert(i, n, c) - вставить n одинаковых символов, равных с. n имеет целочисленный тип, c - char.

S.insert(i, T) - вставить содержимое строки T. T может быть объектом класса string или C-строкой.

S.insert(i, T, pos, count) - вставить символы строки T начиная с символа с индексом pos количеством count.

Подробней о методе insert.

substr

S.substr(pos) - возвращает подстроку данной строки начиная с символа с индексом pos и до конца строки.

S.substr(pos, count) - возвращает подстроку данной строки начиная с символа с индексом pos количеством count или до конца строки, если pos + count > S.size().

Подробней о методе substr.

replace

Заменяет фрагмент строки на несколько равных символов, другую строку или фрагмент другой строки. Способы вызова аналогичны способам вызова метода append, только первыми двумя параметрами являются два числа: pos и count. Из данной строки удаляется count символов, начиная с символа pos, и на их место вставляются новые символы.

S.replace(pos, count, n, c) - вставить n одинаковых символов, равных с. n имеет целочисленный тип, c - char.

S.replace(pos, count, T) - вставить содержимое строки T. T может быть объектом класса string или C-строкой.

S.replace(pos, count, T, pos2, count2) - вставить символы строки T начиная с символа с индексом pos количеством count.

Подробней о методе replace.

find

Ищет в данной строке первое вхождение другой строки str. Возвращается номер первого символа, начиная с которого далее идет подстрока, равная строке str. Если эта строка не найдена, то возвращается константа string::npos (которая равна -1, но при этом является беззнаковой, то есть на самом деле является большим безннаковым положительным числом).

Если задано значение pos, то поиск начинается с позиции pos, то есть возращаемое значение будет не меньше, чем pos. Если значение pos не указано, то считается, что оно равно 0 - поиск осуществляется с начала строки.

S.find(str, pos = 0) - искать первое входение строки str начиная с позиции pos. Если pos не задано - то начиная с начала строки S.

S.find(str, pos, n) - искать в данной строке подстроку, равную первым n символам строки str. Значение pos должно быть задано.

Подробней о методе find.

78. Таблицы кодировок. Пример применения кодов управляющих символов при разработке программ

Таблица кодировки - Это таблица, где каждой букве алфавита (а также цифрам и специальным знакам) присвоен уникальный номер - код символа.

На заре компьютерной эры на каждый символ было отведено по пять бит. Это было связано с малым количеством оперативной памяти на компьютерах тех лет. В эти 32 символа входили только управляющие символы и строчные буквы английского алфавита.

С ростом производительности компьютеров стали появляться таблицы кодировок с большим количеством символов. Первой семибитной кодировкой стала ASCII7. В нее уже вошли прописные буквы английского алфавита, арабские цифры, знаки препинания. Затем на ее базе была разработана ASCII8, в которым уже стало возможным хранение 256 символов: 128 основных и еще столько же расширенных. Первая часть таблицы осталась без изменений, а вторая может иметь различные варианты (каждый имеет свой номер). Эта часть таблицы стала заполняться символами национальных алфавитов.

Но для многих языков (например, арабского, японского, китайского) 256 символов недостаточно, поэтому развитие кодировок продолжалось, что привело к появлению UNICODE.

Управляющие символы (или как их ещё называют — escape-последовательность) — символы которые выталкиваются в поток вывода, с целью форматирования вывода или печати некоторых управляющих знаков С++.

Пример:

#include "stdafx.h"

#include <iostream>

using namespace std;

int main()

{

   cout << "\t\tcontrol characters C++"; // две табуляции и печать сообщения

   cout << "\rcppstudio.com\n"; // возврат каретки на начало строки и печать сообщения

   cout << "\'formatting\' output with \"escape characters\""; // одинарные и двойные ковычки

   cout << "\a\a\a\a\a\a\a\a\a\a\a\a\a\a" <<endl; //звуковой сигнал биппера

   system("pause");

   return 0;

}

Управляющие символы C++ — это не основной способ форматированного вывода, но наиболее простой и наиболее часто используемый.

79. Таблицы кодировок. Пример применения таблиц кодировок при разработке программ с использованием алфавита(англ русск) Анализ диапазонов символов алфавитов. Особенность расположения русского языка

Таблица кодировки - Это таблица, где каждой букве алфавита (а также цифрам и специальным знакам) присвоен уникальный номер - код символа.

На заре компьютерной эры на каждый символ было отведено по пять бит. Это было связано с малым количеством оперативной памяти на компьютерах тех лет. В эти 32 символа входили только управляющие символы и строчные буквы английского алфавита.

С ростом производительности компьютеров стали появляться таблицы кодировок с большим количеством символов. Первой семибитной кодировкой стала ASCII7. В нее уже вошли прописные буквы английского алфавита, арабские цифры, знаки препинания. Затем на ее базе была разработана ASCII8, в которым уже стало возможным хранение 256 символов: 128 основных и еще столько же расширенных. Первая часть таблицы осталась без изменений, а вторая может иметь различные варианты (каждый имеет свой номер). Эта часть таблицы стала заполняться символами национальных алфавитов.

Но для многих языков (например, арабского, японского, китайского) 256 символов недостаточно, поэтому развитие кодировок продолжалось, что привело к появлению UNICODE.

ANAL

80. Таблицы кодировок. Пример преобразования символов в разные регистры

Таблица кодировки - Это таблица, где каждой букве алфавита (а также цифрам и специальным знакам) присвоен уникальный номер - код символа.

На заре компьютерной эры на каждый символ было отведено по пять бит. Это было связано с малым количеством оперативной памяти на компьютерах тех лет. В эти 32 символа входили только управляющие символы и строчные буквы английского алфавита.

С ростом производительности компьютеров стали появляться таблицы кодировок с большим количеством символов. Первой семибитной кодировкой стала ASCII7. В нее уже вошли прописные буквы английского алфавита, арабские цифры, знаки препинания. Затем на ее базе была разработана ASCII8, в которым уже стало возможным хранение 256 символов: 128 основных и еще столько же расширенных. Первая часть таблицы осталась без изменений, а вторая может иметь различные варианты (каждый имеет свой номер). Эта часть таблицы стала заполняться символами национальных алфавитов.

Но для многих языков (например, арабского, японского, китайского) 256 символов недостаточно, поэтому развитие кодировок продолжалось, что привело к появлению UNICODE.

Пример:

#include <iostream>

#include <string>

using namespace std;

int main()

{

string str;

cin >> str;

for(int i=0; i<str.length(); i++){

str[i] = int(str[i]) + 32;

}

cout << str;

return 0;

}

81. Потоки. Высвобождение потока. Функции для С++

Поток — это по сути последовательность инструкций, которые выполняются параллельно с другими потоками. Каждая программа создает по меньшей мере один поток: основной, который запускает функцию main(). Программа, использующая только главный поток, является однопоточной; если добавить один или более потоков, она станет многопоточной. Простыми словами, потоки – это способ сделать несколько вещей одновременно. Класс, дающий возможность создавать потоки с помощью SFML, называется sf::Thread, и вот как это (создание потока) выглядит в действии:

#include <SFML/System.hpp>

#include <iostream>

void func()

{

// эта функция запускается когда вызывается thread.launch()

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

std::cout << "I'm thread number one" << std::endl;

}

int main()

{

// создание потока с функцией func в качестве точки входа

sf::Thread thread(&func);

// запуск потока

thread.launch();

// главные поток продолжает быть запущенным...

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

std::cout << "I'm the main thread" << std::endl;

return 0;

}

В этом коде функции main и func выполняются параллельно после вызова thread.launch(). Результатом этого является то, что текст, выводимый обеими функциями, смешивается в консоли. Точка входа в поток, т.е. функция, которая будет выполняться, когда поток запускается, должна быть передана конструктору sf::Thread. sf::Thread пытается быть гибким и принимать различные точки входа: non-member функции или методы классов, функции с аргументами или без них, функторы и так далее.

sf::Thread thread(&func); thread.launch(); //используется для запуска функции

launch вызывает функцию, которая была передана в конструктор нового потока, и сразу же завершает свою работу, так что вызывающий поток может сразу же продолжить выполнение. Поток автоматически завершает свою работу, когда функция, служащая точкой входа для данного потока, возвращает свое значение. Если вы хотите ждать завершения потока из другого потока, вы можете вызвать его функцию wait.

sf::Thread thread(&func);

// запуск потока

thread.launch();

...

// выполнение блокируется до тех пор, пока поток не завершится

thread.wait();

Функция ожидания также неявно вызывается деструктором sf::Thread, так что поток не может оставаться запущенным (и бесконтрольным) после того, как его экземпляр sf::Thread уничтожается

82. Прямой код. Обратный код. Дополнительный код

Очень часто в вычислениях должны использоваться не только положительные, но и отрицательные числа.Число со знаком в вычислительной технике представляется путем представления старшего разряда числа в качестве знакового. Принято считать, что 0 в знаковом разряде означает знак «плюс» для данного числа, а 1 – знак «минус».

Выполнение арифметических операций над числами с разными знаками представляется для аппаратной части довольно сложной процедурой. В этом случае нужно определить большее по модулю число, произвести вычитание и присвоить разности знак большего по модулю числа.Применение дополнительного кода позволяет выполнить операцию алгебраического суммирования и вычитания на обычном сумматоре. При этом не требуется определения модуля и знака числа.

Прямой код представляет собой одинаковое представление значимой части числа для положительных и отрицательных чисел и отличается только знаковым битом. В прямом коде число 0 имеет два представления «+0» и «–0».Обратный код для положительных чисел имеет тот же вид, что и прямой код, а для отрицательных чисел образуется из прямого кода положительного числа путем инвертирования всех значащих разрядов прямого кода. В обратном коде число 0 также имеет два представления «+0» и «–0».Дополнительный код для положительных чисел имеет тот же вид, что и прямой код, а для отрицательных чисел образуется путем прибавления 1 к обратному коду. Добавление 1 к обратному коду числа 0 дает единое представление числа 0 в дополнительном коде. Однако это приводит к асимметрии диапазонов представления чисел относительно нуля. Так, в восьмиразрядном представлении диапазон изменения чисел с учетом знака.

84. Сортировка пузырьком

Пузырьковая сортировка является одной из самых простых сортировок в программировании, но однако, является очень медленной по отношению к другим видам сортировок. С помощью данной сортировки можно отсортировать массив или вектор.

Принцип работы пузырьковой соритровки:

1.Прохождение по всему массиву

2.Сравнивание между собой пар соседних ячеек

3.Если при сравнении оказывается, что значение предыдущей ячейки больше, чем значение последующей, то значения данных ячеек меняются местами

Для создания пузырьковой сортировки требуется:

1.Создать 2 цикла for, чтобы проходить по всем элементам массива N раз, где N – длина массива

2.Сравнивать ячейки массива с помощью условного ветвления if

3.Менять местами значения ячеек

Исходный код пузырьковой сортировки на языке C++:

/\*

\* Ввести целочисленный массив из N целых чисел.

\* Отсортировать этот массив по возрастанию методом пузырька

\*/

#include <iostream>

using namespace std;

int main()

{

int\* arr; // указатель для выделения памяти под массив

int size; // размер массива

// Ввод количества элементов массива

cout << "n = ";

cin >> size;

if (size <= 0) {

// Размер масива должен быть положитлеьным

cerr << "Invalid size" << endl;

return 1;

}

arr = new int[size]; // выделение памяти под массив

// заполнение массива

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

cout << "arr[" << i << "] = ";

cin >> arr[i];

}

int temp; // временная переменная для обмена элементов местами

// Сортировка массива пузырьком

for (int i = 0; i < size - 1; i++) {

for (int j = 0; j < size - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

// меняем элементы местами

temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

// Вывод отсортированного массива на экран

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

cout << arr[i] << " ";

}

cout << endl;

delete[] arr; // освобождение памяти;

return 0;

}

85. Сортировка вставками

Сортировка вставками (Insertion Sort) — это простой алгоритм сортировки. Суть его заключается в том что, на каждом шаге алгоритма мы берем один из элементов массива, находим позицию для вставки и вставляем. Стоит отметить что массив из 1-го элемента считается отсортированным. Банальным примером сортировки вставками из повседневной жизни является сортировка купюр в кошелке, когда у нас есть 10, 50, 100, 500 рублей и мы кладем 100 рублей между 50 и 500, тем самым выстраивая купюры по возрастанию.

Исходный код сортировки вставками на языке C++:

#include <iostream>

using namespace std;

int i, j, key = 0, temp = 0;

void InsertSort(int\* mas, int n) //сортировка вставками

{

for (i = 0; i < n - 1; i++)

{

key = i + 1;

temp = mas[key];

for (j = i + 1; j > 0; j--)

{

if (temp < mas[j - 1])

{

mas[j] = mas[j - 1];

key = j - 1;

}

}

mas[key] = temp;

}

cout << endl << "Результирующий массив: ";

for (i = 0; i < n; i++) //вывод массива

cout << mas[i] << " ";

}

//главная функция

void main()

{

setlocale(LC\_ALL, "Rus");

int n;

cout << "Количество элементов в массиве > "; cin >> n;

int\* mas = new int[n];

for (i = 0; i < n; i++) //ввод массива

{

cout << i + 1 << " элемент > "; cin >> mas[i];

}

InsertSort(mas, n); //вызов функции

delete[] mas;

system("pause>>void");

}

86. Сортировка методом выбора

Существует множество способов сортировки массивов. Сортировка массивов методом выбора, пожалуй, самая простая для понимания, хотя и одна из самых медленных.

Для сортировки массива методом выбора от наименьшего до наибольшего элемента выполняются следующие шаги:

1.Начиная с элемента под индексом 0, ищем в массиве наименьшее значение.

   2.Найденное значение меняем местами с нулевым элементом.

   3.Повторяем шаги №1 и №2 уже для следующего индекса в массиве (отсортированный элемент больше не трогаем).

Другими словами, мы ищем наименьший элемент в массиве и перемещаем его на первое место. Затем ищем второй наименьший элемент и перемещаем его уже на второе место после первого наименьшего элемента. Этот процесс продолжается до тех пор, пока в массиве не закончатся неотсортированные элементы.

Исходный код сортировки выбором на языке C++:

#include <iostream>

#include <algorithm> // для std::swap. В C++11 используйте заголовок <utility>

int main()

{

const int length = 5;

int array[length] = { 30, 50, 20, 10, 40 };

// Перебираем каждый элемент массива (кроме последнего, он уже будет отсортирован к тому времени, когда мы до него доберемся)

for (int startIndex = 0; startIndex < length - 1; ++startIndex)

{

// В переменной smallestIndex хранится индекс наименьшего значения, которое мы нашли в этой итерации.

// Начинаем с того, что наименьший элемент в этой итерации - это первый элемент (индекс 0)

int smallestIndex = startIndex;

// Затем ищем элемент поменьше в остальной части массива

for (int currentIndex = startIndex + 1; currentIndex < length; ++currentIndex)

{

// Если мы нашли элемент, который меньше нашего наименьшего элемента,

if (array[currentIndex] < array[smallestIndex])

// то запоминаем его

smallestIndex = currentIndex;

}

// smallestIndex теперь наименьший элемент.

// Меняем местами наше начальное наименьшее число с тем, которое мы обнаружили

std::swap(array[startIndex], array[smallestIndex]);

}

// Теперь, когда весь массив отсортирован - выводим его на экран

for (int index = 0; index < length; ++index)

std::cout << array[index] << ' ';

return 0;

}

87. Сортировка Шелла

Идея метода заключается в сравнение разделенных на группы элементов последовательности, находящихся друг от друга на некотором расстоянии. Изначально это расстояние равно d или N/2, где N — общее число элементов. На первом шаге каждая группа включает в себя два элемента расположенных друг от друга на расстоянии N/2; они сравниваются между собой, и, в случае необходимости, меняются местами. На последующих шагах также происходят проверка и обмен, но расстояние d сокращается на d/2, и количество групп, соответственно, уменьшается. Постепенно расстояние между элементами уменьшается, и на d=1 проход по массиву происходит в последний раз. Сортировка Шелла уступает в эффективности быстрой сортировки, но выигрывает у сортировки вставками.

Исходный код сортировки на языке C++:

#include <iostream>

using namespace std;

int i, j, n, d, count;

void Shell(int A[], int n) //сортировка Шелла

{

d = n;

d = d / 2;

while (d > 0)

{

for (i = 0; i < n - d; i++)

{

j = i;

while (j >= 0 && A[j] > A[j + d])

{

count = A[j];

A[j] = A[j + d];

A[j + d] = count;

j--;

}

}

d = d / 2;

}

for (i = 0; i < n; i++) cout << A[i] << " "; //вывод массива

}

//главная функция

void main()

{

setlocale(LC\_ALL, "Rus");

cout << "Размер массива > "; cin >> n;

int\* A = new int[n]; //объявление динамического массива

for (i = 0; i < n; i++) //ввод массива

{

cout << i + 1 << " элемент > "; cin >> A[i];

}

cout << "\nРезультирующий массив: ";

Shell(A, n);

delete[] A; //освобождение памяти

system("pause>>void");

}

88. Линейный поиск

Этот алгоритм перебирает все элементы в массиве, сравнивая их с заданным ключом, из-за полного перебора скорость поиска намного меньше, чем в других алгоритмах. Его обычно используют, если в отрезке поиска находится мало элементов, в ином случае используют другие алгоритмы поиска (один из них бинарный поиск).

Пример работы линейного поиска:

В примере создан массив с 20 элементами и заполнен случайными числами с помощью функции rand.

#include <iostream>

#include <cstdlib>

#include <ctime>

using namespace std;

int main() {

setlocale(LC\_ALL, "rus");

int ans[20]; // создали массив для записи всех индексов

int h = 0;

int arr[20]; // создали массив на 20 элементов

int key; // создали переменную в которой будет находиться ключ

srand(time(NULL));

for (int i = 0; i < 20; i++) {

arr[i] = 1 + rand() % 20; // заполняем случайными числами ячейки

cout << arr[i] << " "; // выводим все ячейки массива

if (i == 9) {

cout << endl;

}

}

cout << endl << endl << "Введите ключ : "; cin >> key; // считываем ключ

for (int i = 0; i < 20; i++) {

if (arr[i] == key) { // проверяем равен ли arr[i] ключу

ans[h++] = i;

}

}

if (h != 0) { // проверяем были ли совпадения

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

cout << "Ключ " << key << " находится в ячейке " << ans[i] << endl; //выводим все индексы

}

}

else {

cout << "Мы не нашли ключ " << key << " в массиве";

}

system("pause");

return 0;

}

Из плюсов у этого алгоритма только одно — простая реализация.

Минус один, но очень большой — код работает медленно.

89. Бинарный поиск

Бинарный поиск — очень быстрый алгоритм с не сложной реализацией, который находит элемент с определенным значением в уже отсортированном массиве. Алгоритм может выдавать неправильный ответ, если массив не был отсортирован.

Пример бинарного поиска на языке C++:

#include <iostream>

#include <algorithm>

using namespace std;

int main() {

setlocale(LC\_ALL, "rus");

int arr[10]; // создали массив на 10 элементов

int key; // создали переменную в которой будет находиться ключ

cout << "Введите 10 чисел для заполнения массива: " << endl;

for (int i = 0; i < 10; i++) {

cin >> arr[i]; // считываем элементы массива

}

sort(arr, arr + 10); // сортируем с помощью функции sort (быстрая сортировка)

cout << endl << "Введите ключ: ";

cin >> key; // считываем ключ

bool flag = false;

int l = 0; // левая граница

int r = 9; // правая граница

int mid;

while ((l <= r) && (flag != true)) {

mid = (l + r) / 2; // считываем срединный индекс отрезка [l,r]

if (arr[mid] == key) flag = true; //проверяем ключ со серединным элементом

if (arr[mid] > key) r = mid - 1; // проверяем, какую часть нужно отбросить

else l = mid + 1;

}

if (flag) cout << "Индекс элемента " << key << " в массиве равен: " << mid;

else cout << "Извините, но такого элемента в массиве нет";

system("pause");

return 0;

}

Плюсы:

 Реализация алгоритма достаточна легкая;

 Быстрая работа алгоритма;

Минусы:

Для поиска массив должен быть упорядочен(отсортирован);

Двоичный поиск должен иметь возможность обратится к любому элементу данных (по индексу). А это значит, что все структуры данных, которые построены на связных списках использоваться не могут;