1. **Ссылочный тип. Выделение и освобождение динамической памяти. Операции над указателями.**

**Указатель**– это статическая переменная, которая предназначена для хранения адреса переменной.

Различают **типизированные** и **нетипизированные** указатели.

**Типизированный указатель** – указатель, который ссылается на область данных определенного типа.

**Схема объявления указателя:**

<имя переменной-указателя>:^<базовый тип данных>;

Выделение и освобождение памяти осуществляется при помощи процедур **New** и **Dispose**.

**New**(<идентификатор указателя>);

**Dispose**(<идентификатор указателя >);

**Нетипизированные указатели**– указатели, базовый тип которых не фиксируется. Такие указатели задают только место, без конкретизации типа значения, содержащегося по этому адресу.

**Схема объявления указателя:**

<имя переменной-указателя>**:**Pointer;

Выделение и освобождение памяти осуществляется при помощи процедур **GetMem** и **FreeMem**.

**GetMem**(<нетипизированный указатель>, <размер памяти в байтах>);

**FreeMem**(<нетипизированный указатель>, <размер памяти в байтах>);

**Над указателями допустимы 3 операции:**

1. Присваивание.
2. Сравнение
3. Разыменование
4. **Программа вывода файла в обратном порядке.**

**program** revprintfile;

**const** n=1000;

**type**

intmas=**array**[1..n] **of** integer;

realmas=**array**[1..n] **of** real;

**var** i,j,k:integer;

p:^intmas;

q:^realmas;

intfile:**file of** integer;

realfile: **file of** real;

**begin**

assign(intfile, 'int.dat');

assign(realfile, 'real.dat');

**new**(p);

reset(intfile);

i:=0;

**while not** eof(intfile) **do**

**begin**

i:=i+1;

read(intfile.p^[i]);

**end**;

i:=p^[i **div** 2+1];

write('значение среднего элемента=',i);

dispose(p);

**new**(q);

reset(realfile);

j:=0;

**while not** eof(realfile) **do**

**begin**

j:=j+1;

read(realfile.q^[j]);

**end**;

**for** k:=j **downto** i **do** writeln(q^[k],' ');

**end**.

1. **Виды связных линейных списков. Описание элемента списка.**

**Односвязный линейный список** – список, в котором предыдущий элемент ссылается на следующий.

**Двусвязный линейный список** – список, в котором предыдущий элемент ссылается на следующий, а следующий на предыдущий.

**Односвязный циклический список** – это односвязный линейный список, в котором последний элемент ссылается на первый.

**Двусвязный циклический список** – это двусвязный линейный список, в котором последний элемент ссылается на первый, а первый – на последний.

**Стек** – односвязный список, в котором компоненты добавляются и удаляются только со стороны вершины списка.

**Очередь** – односвязный список, в котором компоненты добавляются в конец, а удаляются со стороны вершины списка.

Для **задания списковых структур** необходимо определить элемент списка в виде записи, в состав которой входит информационное поле и поле-указатель на следующий элемент.

**type**

ukaz=^element;

element=**record**

inf:<тип информационного поля>;

next:ukaz;

**end**;

1. **Стек. Программа формирования и обработки.**

**Стек** – односвязный список, в котором компоненты добавляются и удаляются только со стороны вершины списка.

**type procedure** del(**var** top:ukaz);

ukaz=^element; **begin**

element=**record if** top<>nil **then**

inf:integer; **begin**

next:ukaz; temp:=top;

**end**; top:=top^.next;

**var** top,temp:ukaz; dispose(temp);

v:integer; **end end;**

**procedure** dobs(**var** top:ukaz); **procedure** prosmotr(**var** top:ukaz)

**begin begin**

read(v); temp:=top;

**new**(temp); **while** temp<>nil **do**

temp^.inf:=v; **begin**

temp^.next:=top; writeln(temp^.inf);

top:=temp; temp:=temp^.next;

**end**; **end end;**

1. **Очередь. Программа формирования и обработки.**

**Очередь** – односвязный список, в котором компоненты добавляются в конец, а удаляются со стороны вершины списка.

**type procedure** dobs(**var** left,right:ukaz);

ukaz=^element; **begin**

element=**record** read(v);

inf:integer; **new**(temp);

next:ukaz; temp^.inf:=v;

**end**; temp^.next:=nil;

**var** left,right,temp:ukaz; right:=temp;

v:integer; left:=temp;

**end**;

**procedure** dob\_v\_konec(**var** left,right:ukaz);

**begin**

read(v);

**new**(temp);

temp^.inf:=v;

temp^.next:=nil;

**if** left=nil **then** left:=temp

**else**

**begin**

right:=left;

**while**(right^.next<>nil) **do** right:=right^.next;

right^.next:=temp;

right:=temp;

**end**;

**end**;

**procedure** prosmotr(**var** left:ukaz); **procedure** del(**var** left:ukaz);

**begin begin**

temp:=left; **if** left<>nil **then**

**while** temp<>nil **do begin**

**begin** temp:=left;

writeln(temp^.inf); left:=left^.next;

temp:=temp^.next; dispose(temp);

**end if** left=nil **then** right:=nil;

**end**; **end** **end**;

1. **Двунаправленные списки. Удаление элемента двунаправленного списка.**

**Двусвязный линейный список** – список, в котором предыдущий элемент ссылается на следующий, а следующий на предыдущий.

**procedure** Delete(top: ukaz; del\_value: integer);

**var** element, Right, Left: ukaz;

**begin**

**if** top = nil **then** writeln('List not init')

**else**

**while** (top <> nil) **do**

**begin**

**if**(top^.value = del\_value) **then**

**begin**

element:= top;

Right:= element^.next;

Left:= element^.prev;

dispose(element);

Right^.prev:= Left;

Left^.next:= Right;

**end else**

top:= top^.next;

**end**; **end**;

1. **Добавление элемента в двунаправленный список.**

**procedure** dobs(**var** left,right:ukaz);

**begin**

read(v);

**new**(temp);

temp^.inf:=v;

temp^.nextr:=nil;

temp^.nextl:=nil;

right:=temp;

left:=temp;

**end**;

1. **Двунаправленные списки. Программа формирования и обработки.** 6) 7)
2. **Бинарные деревья. Процедуры обхода.** 11)
3. **Процедура добавления узла в бинарное дерево.**

**function** addtree(top:treeptr; newnode:string):treeptr;

**begin**

**if** top=nil **then**

**begin**

**new**(top);

top^.data:=newnode;

top^.left:=nil;

top^.right:=nil;

**end**

**else**

**if** top^.data>newnode **then** top^.left:=addtree(top^.left, newnode)

**else** top^.right:=addtree(top^.right, newnode)

**end**.

1. **Процедуры обхода бинарного дерева и поиска.**

**procedure** lr(top:treeptr);

**begin**

**if** top<>nil **then**

**begin**

lr(top^.left);

writeln(top^.data);

lr(top^.right);

**end**

**end;**

**procedure** rl(top:treeptr);

**begin**

**if** top<>nil **then**

**begin**

lr(top^. right);

writeln(top^.data);

lr(top^.left);

**end**

**end;**

**procedure** Verh\_vniz(top:treeptr);

**begin**

**if** top<>nil **then**

**begin**

writeln(top^.data)

lr(top^.left);

lr(top^.right);

**end**

**end;**

**function** search(root:treeptr; node:string):treeptr;

**var** found:boolean;

**begin**

found:=false;

**while** (root<>nil) **and** (**not** found) **do**

**if** root^.data=node **then** found:=true

**else**

**if** root^.data>node **then** root:=root^,left

**else** root:=root^.right;

search:=root;

**end**;

1. **Программа поиска с включением узлов в бинарное дерево.**

**type** treeptr=^treenode;

treenode=**record**

data:string;

count:integer;

left,right:treeptr;

**end**;

**var** root:treeptr;

n:integer;

key:string;

**procedure** printtree(top:treeptr; otstup:integer);

**begin**

**if** top<>nil **then**

**with** top^ **do begin**

printtree(right, otstup+1);

writeln(' ':otstup, data);

printtree(left, otstup+1);

**end**;

**end**;

**procedure** searchl(**var** top:treeptr; node:string);

**begin**

**if** top=nil **then begin**

**new**(top);

**with** top^ **do begin**

data:=node;

count:=1;

left:=nil;

right:=nil;

**end**;

**end**

**else**

**if** node<top^.data **then** searchl(top^.left, node)

**else if** node>top^.data **then** searchl(top^.right, node)

**else** top^.count:=top^.count+1;

**end**;

**begin**

root:=nil;

readln(n);

**while** n>0 **do**

**begin**

readln(key);

searchl(root, key);

n:=n-1;

**end**;

printtree(root, 0);

**end**.

1. **Удаление узла из бинарного дерева поиска.**

**procedure** delete(**var** top:treeptr; node:integer);

**var** q:treeptr;

**begin**

**if** top=nil **then exit**

**else if** node<top^.data **then** delete(top^.l.node)

**else if** node>top^.data **then** delete(top^.r.node)

**else begin**

q:=top;

**if** q^.r=nil **then** top:=q^.l

**else if** q^.l=nil **then** top:=q^.r

**else** delr(q^.l);

dispose(q);

**end**; **end**;

1. **Методология структурного программирования.**

**Теорема Бёма** - **Якопини** — положение структурного программирования, согласно которому любой исполняемый алгоритм может быть преобразован к структурированному виду, то есть такому виду, когда ход его выполнения определяется только при помощи трёх структур управления: последовательной, ветвлений и циклов.

1. **Использование модулей в языке Паскаль.**

**Модуль Паскаля** — это автономно компилируемая программная единица, включающая в себя различные компоненты раздела описаний (типы, константы, переменные, процедуры и функции) и, возможно, некоторые исполняемые операторы инициирующей части.

**Структура модуля:**

**Unit** <имя\_модуля>;  
**interface**<интерфейсная часть>;  
**implementation**<исполняемая часть>;  
**begin**  
 <инициирующая часть>;   
**end**.

**Заголовок модуля** Паскаля состоит из зарезервированного слова **unit** и следующего за ним имени модуля.

Для правильной работы среды Турбо Паскаля и возможности подключения средств, облегчающих разработку больших программ, имя модуля Паскаля должно совпадать с именем дискового файла, в который помещается исходный текст модуля.

Имя модуля Паскаля служит для его связи с другими модулями и основной программой. Эта связь устанавливается специальным предложением:

**uses** <список модулей>

**Интерфейсная часть** открывается зарезервированным словом **interface**. В этой части содержатся объявления всех глобальных объектов модуля (типов, констант, переменных и подпрограмм), которые должны быть доступны основной программе.

**Исполняемая часть** модуля Паскаля начинается зарезервированным словом **implementation** и содержит описания подпрограмм, объявленных в интерфейсной части.

В **инициирующей части** размещаются исполняемые операторы, содержащие некоторый фрагмент программы. Эти операторы исполняются до передачи управления основной программе и обычно используются для подготовки ее работы. Например, в инициирующей части могут инициироваться переменные, открываться файлы и т.д.

Стандартные модули: SYSTEM, DOS, CRT, GRAPH, OVERLAY, TURBO3, GRAPH3.

1. **Программа печати таблицы температур (примеры с циклами while и for).**

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

main() { main() {

int farh, celsius; int farh;

int lower, upper, step; int lower, upper, step;

lower=0; //нижняя граница температур lower=0; // нижняя граница температур upper=300; //верхняя граница температур upper=300; //верхняя граница температур

step=20; //шаг step=20; //шаг

farh=lower; for (farh=lower; farh<=upper; farh=farh+step)

while (farh<=upper) { printf("%3d %6.1d\n", farh, 5.0/9.0\*(farh-32));

celsius=5\*(farh-32)/9; }

printf("%d\t %d\n", farh, celsius);

farh=farh+step;

}

}

1. **Ввод-вывод литер. Программы подсчета литер и строк входного потока.**

**Подсчёт строк: Подсчёт литер:**

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

main() { main() {

int c, nl; long nc;

nl=0; nc=0;

while ((c=getchar())!=EOF) while (getchar()!=EOF) ++nc;

if (c=='\n') ++nl; printf("%ld\n", nc);

printf("%d\n", nl); }

}

1. **Программа подсчёта слов во входном потоке.**

**Подсчёт слов:**

#include <stdio.h>

#define IN 1

#define OUT 0

main() {

int c, nl, nw, nc, state;

state=OUT;

nl=nw=nc=0;

while ((c=getchar())!=EOF)

{

++nc;

if (c=='\n') ++nl;

if (c==' '||c=='\n'||c=='\t') state=OUT;

else if (state==OUT)

{

state=IN;

++nw;

}

}

printf("%d %d %d \n", nl, nw, nc);

}

1. **Программа подсчёта цифр, пробельных и прочих литер.**

#include <stdio.h>

main() {

int c, i, nwhite, nother;

int ndigit[10];

nwhite=nother=0;

for(i=0;i<10;++i) ndigit[i]=0;

while ((c=getchar())!=EOF)

if (c>='0'&&c<='9') ++ndigit[c-'0']

else if (c==' '||c=='\n'||c=='\t') ++nwhite

else ++nother;

printf("цифры=");

for (i=0;i<1;++i) printf ("%d", ndigit[i]);

printf ("\n пробелы=%d, другие=%d\n", nwhite, nother);

}

1. **Программа чтения строк и печать самой длинной строки.**

#include <stdio.h>

#define MAXLINE 1000

int getline(char line[], int maxline);

void copy(char to[], from[]);

main() {

int len, max;

char line[MAXLINE], longest[MAXLINE];

max=0;

while((len=getline(line, MAXLINE))>0)

if (len>max)

{

max=len;

copy(longest, line);

}

if (max>0) printf("%s", longest);

}

int getline(char s[], int lim) {

int c, i;

for (i=0; i<lim-1 && (c=getchar())!=EOF && c!='\n'; ++i) s[i]=c;

if (c=='\n') {s[i]=c; ++i}

s[i]='\0';

return i;

}

1. **Внешние переменные и область видимости.**

**Область видимости переменных** — это те части программы, в которой пользователь может изменять или использовать переменные в своих нуждах.

**Глобальными переменными** называются те переменные, которые были созданы вне тела какого-то блока. Их можно всегда использовать во всей программе, вплоть до ее окончания работы.

**Локальные переменные** — это переменные созданные в блоках.

1. **Базовые типы и размеры данных.**

**В Си существует несколько базовых типов:**

1. **char – 1 байт**
2. **int – 2 или 4 байта**
3. **float – 4 байта**
4. **double – 8 байтов**

**Модификаторы:**

1. **short (короткие)**
2. **long (длинный)**
3. **signed (со знаком)**
4. **unsigned (без знака)**

**Типы с модификатором:**

1. **signed char (-128..127)**
2. unsigned char (0..255)
3. short int (-32768..32767)
4. long int (-2147483648..2147483647)
5. unsigned int (0..65535)
6. **Константы (числовые, символьные, строковые, перечисления).**

**Числовые константы:**

1. Константа типа **long** завершается буквой **l** или **L**.
2. Беззнаковые (**unsigned**) константы заканчиваются буквой **u** или **U**.
3. **Unsigned long** – **ul** или **UL**.
4. **Float** – **f** или **F**.
5. **Целое значение** может иметь десятичное, восьмеричное и шестнадцатеричное представление.

**Символьные константы:**

Записывается в виде символа, заключенного в одиночные кавычки. Значением символьной константы является числовой код символа.

Некоторые символы записываются в виде определенной последовательности. Например ‘\n’.

**Строковые константы:**

Строковые константы – ноль или более символов, заключенных в двойные кавычки. В строки можно включать определенные последовательности (‘\n’ и т.д.). Строковые константы можно склеивать.

Фактические строковая константа – это массив символов, поэтому в конце присутствует символ ‘\0’.

**Константы перечисления:**

enum <имя> {<список констант>}

Первое имя имеет значение 0, следующее – 1 и т.д. Если не все значения специфицированы, то они продолжают свою прогрессию, начинаю от последнего специфицированного значения.

1. **Декларации (объявления). Классы памяти.**

Все переменные должны быть объявлены раньше, чем они будут использоваться в программе. Объявление содержит список из одной или несколько переменных этого типа.

Например: int lower, upper, step;

В своём объявлении переменная может быть инициализирована.

Например: int i=0;

К любой переменное может быть применен модификатор const.

Например: const int i=2;

**Общий вид оператора объявления:**

[класс памяти] [const] <тип> <имя> [инициализатор]

**Классы памяти:**

1. Auto – автоматическая переменная. Память под неё выделяется в стеке и инициализируется каждый раз при выполнении оператора, содержащего ее определение. Освобождение памяти происходит при выходе из блока, в котором описана переменная. Время её жизни – с момента описания до конца блока.
2. Extern – означает, что переменная является внешней и определяется в другом месте программы.
3. Static – статистическая переменная. Существует в течение всего времени выполнения программы. Инициализируется однократно при первом выполнении оператора, содержащего определение переменной.
4. Register – аналогично auto, но память выделяется по возможности в регистрах процессора.
5. **Преобразования типов при вычислении выражений.**

В операторах и выражениях должны использоваться переменные и константы только одного типа. Если это не так, то компилятор использует набор правил для автоматического преобразования типов:

1. Если операция выполняется над данными двух различных типов, обе величины приводятся к "высшему" из двух типов.
2. Последовательность имен типов, упорядоченных от "высшего" к "низшему", выглядит так: **double**, **float**, **long**, **int**, **short** и **char**. Применение ключевого слова **unsigned** повышает ранг соответствующего типа данных со знаком.
3. **Преобразования типов при присвоениях.**

Для операции присваивания результат вычисления выражения правой части приводится к типу переменной, которой должно быть присвоено это значение. При этом может произойти повышение типа, либо его понижение.

1. **Инкрементные и декрементные операторы. Побитовые операторы.**

**Инкремент** – увеличение на 1.

**Декремент** – уменьшение на 1.

++<имя> - префиксные инкремент –– сначала увеличивается, затем используется

––<имя> - префиксные декремент –– сначала уменьшается, затем используется

<имя>++ - постфиксный инкремент –– сначала используется, затем увеличивается

<имя>–– - постфиксные декремент –– сначала используется, затем уменьшается

**Побитовые операторы:**

1. & – оператор побитового И сравнивает каждый бит первого операнда с соответствующим битом второго операнда. Если оба бита равны 1, соответствующий бит результата устанавливается равным единице, в противном случае — нулю.
2. | – оператор побитового включающего ИЛИ сравнивает каждый бит первого операнда с соответствующим битом второго операнда. Если любой из битов равен единице, соответствующий бит результата устанавливается равным единице, а в противном случае — нулю.
3. ^ – ставит 1, если соответствующие разряды операндов имеют различные значения, и 0, когда они совпадают.
4. << – сдвиг влево, >> – сдвиг вправо. Сдвигают свой левый операнд на число битовых позиций, задаваемое правым операндом, который должен быть не отрицательным.
5. ~ – побитовое отрицание. Превращает каждый единичный байт в нулевой и наоборот.
6. **Операторы присваивания.**

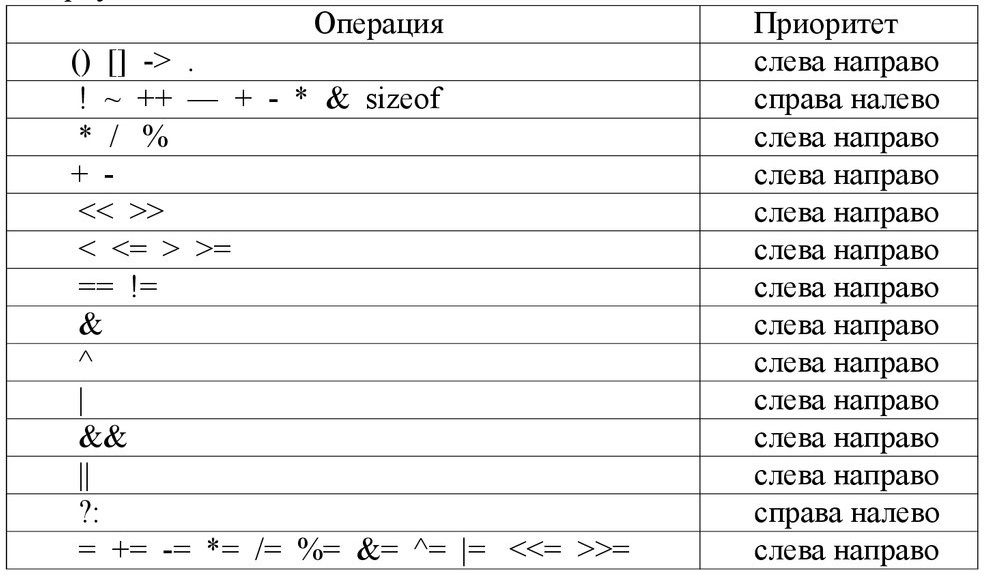
**Знак операции присваивания**: =

**Операторы присваивания:**

1. +=
2. -=
3. \*=
4. /=
5. %=
6. **Условные выражения. Приоритет и порядок вычисления.**

**Тернарная условная операция** имеет 3 аргумента и возвращает свой второй или третий операнд в зависимости от значения логического выражения, заданного первым операндом. Синтаксис тернарной операции в языке Си:

<условие> ? <выражение1> : <выражение2>



1. **Инструкция if-else. Программа бинарного поиска.**

if (<выражение>) <оператор1>; [else <оператор2>]

**Программа бинарного поиска:**

int binsearch(int x, int v[], int n)

{

int low, high, mid;

low=0;

high=n-1;

while(low<=high)

{

if (x<v[mid]) high=mid-1;

else if (x>v[mid]) low=mid+1;

else return mid;

}

return -1;

}

1. **Переключатель switch. Программа подсчёта цифр, пробелов и прочих литер.**

switch(<выражение>) {

case <константное выражение\_1>:[<операторы\_1>];

case < константное выражение \_2>:[<операторы \_2>];

case < константное выражение \_n>:[<операторы \_n>];

[<default>:< операторы >];

}

**Программа подсчёта цифр, пробелов и прочих литер:**

#include <stdio.h>

main() {

int c, i, nwhite, ndigit[10];

nwhite=nother=0;

for(i=0; i<10; i++) ndigit[i]=0;

while ((c=getchar())!=EOF)

{

switch(c)

{

case '0': case '1': case '2': case '3': case '4':

case '5': case '6': case '7': case '8': case '9':

ndigit[c-'0']++; break;

case '': case '\n': case '\t': nwhite++; break;

default: nother++;

}

}

printf ("цифры=");

for (i=0; i<10; i++) printf ("%d", ndigit[i]);

printf (", пробелов= %d, прочих = %d\n", nwhite, nother);

return 0;

}

1. **Циклы while и for. Определение функции atoi.**

while (<условие>) { <тело цикла> }

for (<начало>;<условие>;<шаг>) { <тело цикла> }

**Функция atoi:**

int atoi (const char \* string);

Функция **atoi** преобразует строку **string** в целое значение типа **int.** Строка должна содержать корректную запись целого числа. В противном случае возвращается 0.

1. **Цикл do-while. Определение функции itoa.**

do <тело цикла>; while (<выражение>);

**Функция itoa:**

char \*itoa (int value, char \* string, int radix);

Функция **itoa** преобразует целое число **value** в строку string в формате **radix**.

1. **Инструкции break, continue, goto.**
2. Инструкция **break** используется для выхода из циклов. Если есть несколько вложенных циклов, то произойдёт выход из текущего цикла.
3. Оператор **continue**используется только в циклах. В операторах for, while, do while**,**оператор **continue**выполняет пропуск оставшейся части кода тела цикла и переходит к следующей итерации цикла.
4. С помощью инструкции **goto** можно перейти на указанную метку. После ключевого слова **goto** необходимо указать имя метки, с которой будет продолжено выполнения программы. Сама метка ставится при помощи ключевого слова **label**.
5. **Объявление и определение функций. Пример: определение функций atof.**

**Функция** — это группа операторов, вызываемая по имени и возвращающая в точку вызова предписанное значение.

**Формат заголовка функции**: <тип> <имя> (<список формальных параметров>)

Для **вызова** **функции** указывают ее имя, а также передают ей значения фактических параметров: <имя функции> (<список фактических параметров>);

**Определение функций atof:**

double atof (const char \*string);

Функция **atof** преобразует строку в число типа double. Преобразование заканчивается, когда встречается символ не являющейся цифрой, точкой или когда будет преобразована вся строка. Если первый символ строки отличается от перечисленных выше символов, то функция вернет 0 и завершит работу.

1. **Рекурсивные функции. Пример: быстрая сортировка массива.**

В Си допускается рекурсивное обращение к функциям, т.е. функция может обращаться сама к себе, прямо или косвенно.

**Программа быстрой сортировки массива:**

void qsort (int v[], int left, int right) { void swap (int v[], int i, int j) {

int i, last; int temp;

void swap (int v[], int i, int j); temp=v[i];

if (left>=right) return; v[i]=v[j];

swap (v, left, (left+right)/2); v[j]=temp;

last=left; }

for (i=left+1; i<=right; i++)

if (v[i]<v[left]) swap (v, ++last, i);

swap (v, left, last);

qsort (v, left, last-1);

qsort (v, last+1, right);

}

1. **Директива #include.**

**Препроцессором** называется первая фаза работы компилятора. Инструкции препроцессора называются директивами. Они должны начинаться с символа #.

**Директива #include** предлагает компилятору включить другой исходный файл, имя которого ука­зывается после директивы. Имя файла заключается в двойные кавычки или в <>.

Если подключаемый файл указан в <>, то поиск будет происходить в стандартных каталогах, предназначенных для хранения заголовочных файлов. В случае, если подключаемый файл заклю­чен в двойные кавычки, поиск будет происходить в текущем рабочем каталоге. Если файл не найден, то поиск продолжается в стандартных каталогах.

1. **Директива #define.**

**Директива #define** определяет идентификатор и последовательность символов, которой будет за­мещаться данный идентификатор при его обнаружении в тексте программы.

1. **Указатели. Управление памятью.**

**Указатель**– это статическая переменная, которая предназначена для хранения адреса переменной.

В общем случае переменная типа указатель объявляется следующим образом:

<тип> \*<переменная\_указатель>;

**Управление памятью:**

1. new
2. delete
3. malloc
4. free
5. **Указатели и аргументы функций.**

Поскольку в Си функции в качестве своих аргументов получают значения параметров, нет прямой возможности, находясь в вызванной функции, изменить переменную вызывающей функции.

swap(a, b);

Чтобы получить желаемый эффект, вызывающей программе надо передать указатели на те значения, которые должны быть изменены.

swap(&a, &b);

В самой же функции параметры должны быть объявлены как указатели:

void swap(int \*px, int \*py)

1. **Указатели и массивы.**

Если до начала работы программы количество элементов в массиве неизвестно, то следует использовать динамические массивы. Память под них выделяется во время выполнения программы с помощью оператора **new** или функции **malloc**, а освобождается с помощью оператора **delete** или функции **free** соответственно, например:

#include <stdlib.h>

main () {

int n;

cout<<"Введите размерность массива", cin>>n;

int \*a=new int[n];

double \*b=(double\*) malloc (n\*sizeof(double));

//...............

delete []a;

free (b);

return 0;

}

1. **Адресная арифметика.**

int \*ptr1, \*ptr2, j, \*\*dptr;

1. Операция присваивания: ptr1=&j;
2. Операция взятия адреса: dptr=&ptr1;
3. Операция разыменования: j=\*ptr2;
4. Сложение с целым: ptr2=ptr1+j;
5. Разность указателей: j=ptr1-ptr2;
6. Операция индексации: ptr1[j]=10;
7. **Символьные указатели. Определение функции ctrcpy, strcmp.**

Указатель на строку символов можно объявить следующим образом:

char \*pmessage;

Присваивание pmessage=”now is the time”; поместит в pmessage указатель на символьный массив, при этом сама строка не копируется, копируется лишь указатель на неё.

**char\* strcpy(s,ct)** копирует стринг **ct** в стринг **s**; возвращает **s.**

void strcpy (char \*s, char \*t)

{

while ((\*s++=\*t++)!='\0');

}

**char\* strcmp(cs,ct)** сравнивает **cs** и **ct.**

int strcmp (char \*s, char \*t) {

for (; \*s==\*t; s++, t++)

if (\*s=='\0') return 0;

return \*s-\*t;

}

1. **Массивы указателей. Программа быстрой сортировки строк.**

Можно создавать массивы указателей. Для объявления массива целочисленных указателей из десяти элементов следует написать:

int \*x[10];

**Программа быстрой сортировки строк:**

void qsort (char \*v[], int left, int right) { void swap (char \*v[], int i, int j) {

int i, last; char \*temp;

void swap (char \*v[], int i, int j); temp=v[i];

if (left>=right) return; v[i]=v[j];

swap (v, left, (left+right)/2); v[j]=temp;

last=left; }

for (i=left+1; i<=right; i++)

if (strcmp(v[i],v[left])<0) swap (v, ++last, i);

swap (v, left, last);

qsort (v, left, last-1);

qsort (v, last+1, right);

}

1. **Многомерные массивы.**

**Многомерный массив** – массив, в котором положение элемента определяется несколькими индексами.

int a[2][3];

В памяти такой массив располагается построчно.

1. **Аргументы в командной строке.**

В момент вызова **main** получает два аргумента. В первом, обычно называемом **argc**, содержится количество аргументов, задаваемых в командной строке. Второй **argv** является указателем на массив литерных указателей.

1. **Программа печати строк по образцу.**

1. **Указатели на функции. Пример: быстрая сортировка строк.**

**Указатель на функцию** мы можем определять в виде отдельной переменной с помощью следующего синтаксиса:

тип (\*имя\_указателя) (параметры);

**Например:**

int add(int x, int y)

{

return x+y;

}

int subtract(int x, int y)

{

return x-y;

}

int main(void) {

int a = 10;

int b = 5;

int result;

int (\*operation) (int a, int b);

operation=add;

result = operation(a, b);

printf("result=%d \n", result); // result=15

operation = subtract;

result = operation(a, b);

printf("result=%d \n", result); // result=5

return 0;

}

1. **Стандартный ввод-вывод.**
2. Функция **getchar()** – чтение одного символа из стандартного потока ввода.
3. Функция **putchar()** – вывод одного символа в стандартный поток вывода.
4. **Форматный ввод-вывод.**
5. Функция **printf** преобразует, форматирует и печатает свои аргументы в стандартном потоке вывода под управлением строки формата.

Строка формата содержит два вида объектов: обычные символы, которые копируются в выходной поток, и спецификации преобразования, каждая из которых вызывает преобразование и печать очередного аргумента.

**Спецификация преобразования:**

%[флаг][ширина][.точность][h|l]символ\_формата

1. Функция **scanf** читает символы из стандартного потока и рассылает результаты в свои остальные аргументы. Перед аргументов надо поставить амперсант(&).
2. **Доступ к файлам.**

Открытие файла при помощи функции **fopen,** которая имеет следующий прототип:

FILE \*fopen (const char \*filename, const char \*mode);

Параметр mode указывает режим открытия файлов (r, w, a, r+, w+, a+).

Для завершения работы с файлов он должен быть закрыт с помощью функции **fclose**.

1. **Форматированный и неформатированный ввод-вывод файлов.**
2. **Форматированный ввод-вывод** текстовых файлов организуется с помощью функций **fscanf** и **fprintf**. Эти функции аналогичны функциям **scanf** и **printf** с той лишь разницей, что их первым аргументов является указатель на файл.
3. Для осуществления **неформатированного ввода-вывода** применяются **fread** и **fwrite**. Функция **fread** считывает блоки данных из файлового потока в буфер. А функция **fwrite** выполняет обратную операцию, то есть записывает блоки данных из буфера в файловый поток.
4. **Структуры.**

Описание **структуры** начинается с ключевого слова **struct** и содержит список описаний полей в фигурных скобках. Поля структуры могут иметь любой тип, кроме типа этой структуры, но могут быть указателями на него.

struct [<имя структуры>] {

<тип1> <поле1>;

<тип2> <поле2>;

...

<тип n> <поле n>;

} [список переменных];

Для доступа к полям структуры используется **операция выбора**, обозначаемая точкой:

<имя структурной переменной>.<имя поля>

1. **Массивы структур. Программа сравнения двух дробей.**

#include <stdio.h>

struct drob {

int ch, zn;

} d1, d2;

int main () {

printf ("Числитель и знаменатель 1 дроби:\n"); scanf ("%d %d", &d1.ch, &d1.zn);

printf ("Числитель и знаменатель 2 дроби:\n"); scanf ("%d %d", &d2.ch, &d2.zn);

if (d1.ch\*d2.zn==d2.ch\*d1.zn) printf("=");

else if (d1.ch\*d2.zn>d2.ch\*d1.zn) printf ("1>2");

else if (d1.ch\*d2.zn<d2.ch\*d1.zn) printf ("2>1");

return 0;

}

1. **Структуры со ссылками на себя.**

Элементов структуры может быть указатель, который указывает на элемент своего собственного типа данных:

struct list {

int data;

struct list \*ukaz;

};

1. **Бинарные деревья.**

В бинарном дереве каждый узел содержит указатель на левое и правое поддерево. Поэтому узел дерева представляют в виде структуры с тремя полями:

struct tree {

int data;

struct tree \*left;

struct tree \*riqht;

};