**1.1 Арифметические операторы.**

Основные арифметические операторы: **сложения, вычитания, умножения, деления.** В языке С++ применяются для представления арифметических действий. В С++ нет оператора возведения в степень.

**+ (сложение): +x, x+y  
- (вычитание): -x, x-y  
\* (умножение): x\*y  
/ (деление): x/y  
% ( остаток от деления): x%y**

**Существуют префиксные и постфиксные  операторы.**   
Префиксные (до) указывают C++ сначала увеличить (или уменьшить) значение переменной, а затем использовать это значение.   
Постфиксные (после) указывают C++ сначала использовать значение переменной, а затем увеличить (или уменьшить) его.

**1.2. Побитовые операторы.**К побитовым, или поразрядным операторам относятся:   
**операторы поразрядного И** (побитовое умножение) **&;**   
**операторы поразрядного ИЛИ** (побитовое сложение) **|**;   
**операторы поразрядного исключающего ИЛИ ^;**   
унарный оператор поразрядного отрицания **~** .   
оператор сдвига влево <<;  
оператор сдвига вправо >>;

Операнды поразрядных операций могут быть любого **целого** типа.

**Оператор &** часто используется для маскирования некоторого множества битов.

**Оператор |** используется для включения битов

Используя **оператор ^** можно обменять значения двух переменных, не используя третью: x=x^y, y=x^y, x=x^y

**Оператор << (сдвиг влево) выполняет** побитовый сдвиг влево левого операнда на количество разрядов, соответствующее значению правого операнда. Сдвиг на 1 бит эквивалентный умножению на 2. Результатом является целое число.

**Оператор >> (сдвиг вправо):** выполняет побитовый сдвиг вправо левого операнда на количество разрядов, соответствующее значению правого операнда. Если число без знака, то левые биты = 0. Сдвиг на 1 бит эквиалентный делению на 2. Результатом является целое число.

Сдвиг вправо может быть арифметическим (т. е. освобождающиеся слева разряды заполняются значениями знакового разряда) или логическим в зависимости от реализации, однако гарантируется, что сдвиг вправо целых чисел без знака будет логическим и освобождающиеся слева разряды будут заполняться

**1.3.** **Логические операторы**

В качестве операндов выступают логические значения, результат – тоже логическое значение  
! (отрицание)

|| (логическое “или”)   
&& (логическое “и”)

**1.4. Операторы сравнения (отношения).**Любое значение, не равное нулю, - истина. Если тождественно равное нулю – ложь.Операторы сравнения и логические операторы не изменяют значения своих операндов, а только вычисляют значение: 0 (false), 1 (true).

Сравнивать можно операнды любого типа, но

либо они должны быть оба одного и того же встроенного типа (сравнение на равенство и неравенство работает для двух величин любого типа),   
 либо между ними должна быть определена соответствующая операция сравнения. Результат – логическое значение true или false.

**1.5.** **Приоритет операторов** определяется круглыми скобками. В противном случае предполагается, что операторы имеют следующие относительные приоритеты.

**! -(изменение знака) + + + - -(унарные)  
\* / %  
+ -  
<< >>  
< <= > >=        = = !=  
&  
^  
|**

**&&  
||  
? : (условный оператор)**

**1.6. Выражения**

Программа оперирует с данными. Числа можно складывать, вычитать, умножать, делить. Из разных величин можно составлять выражения, результат вычисления которых – новая величина.

Выражение, после которого стоит точка с запятой – это оператор-выражение. Его смысл состоит в том, что компьютер должен выполнить все действия, записанные в данном выражении, иначе говоря, вычислить выражение.

**Выражения** – это переменные, функции и константы, называемые операндами, объединенные знаками операций. Операции могут быть унарными , например, минус; могут быть бинарными. Если в выражении встречаются переменные и константы разных типов, то все они приводятся к типу с наибольшим диапазоном значений.

**1.7.** **Преобразование типов**

Если в выражении встречаются переменные и константы разных типов, то все они приводятся к типу с наибольшим диапазоном значений.

Для явного преобразования типа выражения используется оператор:  
(<тип>) <выражение>   
где в скобках – один из простых типов данных.  
(int) (1.5 / 0.3); // тип int  
(float) (1.5 / 0.3); // тип float

В выражении приоритет преобразования типов данных приравнивается к приоритету унарных операторов.

**2.1. Инструкция break**

Прекращает работу блока switch и циклов.

**2.2. Инструкция continue.**

Инструкция continue вынуждает ближайший содержащий ее цикл (**for, while** или **do-while**) начать следующий шаг итерации. Операторы, стоящие в цикле после continue, не выполняются.

**2.3. Инструкция goto.**

Инструкция goto позволяет реализовать передачу программного управления из одной точки программы в другую, отмеченную *мет­кой перехода*. Метка перехода состоит из идентификатора и за­вершающего двоеточия, как и метки **switch**-оператора.

Конструкция имеет синтаксис: goto*метка****;***

**2.4. Выход из программы exit()**

С помощью функции ехit() можно прервать выполнение программы в любом месте. Такой вы­ход из программы исполь­зуется при возникновении серьезной ошибки, которая делает дальней­шее выполнение программы бессмысленным или невоз­можным.

**2.5. Инструкция if …else.**

В некоторых ситуациях необходимо указать не только оператор (блок операторов), выполняемый в случае получения результата **true** при вычислении условия, но и оператор (блок операторов), выполняемый при получении результата **false.**

Синтаксис инструкции выбора в этом случае будет следующим:   
 **if (условие) оператор;  
 else оператор;**

В качестве *условия* может использоваться любое выражение арифметического или приводимого к нему типа (например, целое число, арифметическое выражение, логическое выражение, выражение сравнения или вызов функции с соответствующим типом возвращаемого значения).

В качестве *оператора* может выступать отдельный оператор или блок операторов. Отдельный оператор всегда должен заканчиваться точкой с запятой (ограничителем операторов). Если используется блок операторов, то он должен быть заключен в фигурные скобки.

**2.6. Вложение else if**

С помощью ключевых слов if и else можно составлять так называемые else-if-конструкции, которые могут осуществить проверку сразу нескольких выражений, не используя сложные условия .   
if *(условие) оператор;*  
else if *(условие) оператор;*   
 else if *(условие) оператор;*   
*…* else оператор;

Условия проверяются в той последовательности, в которой они перечислены в программном коде.

Если результатом одного из условий является true (1),то выполняется следующий за условием оператор и проверка оставшихся **elseif**-условий не осуществляется.

Если ни одно из проверенных условий не дало в результате значения **true,** то выполняются операторы, относящиеся к последнему **else.**

Последняя **else**-конструкция является необязательной, следовательно, она может отсутствовать, если в программе не предусмотрено выполнения какого-либо действия по умолчанию.

**2.7. Инструкция switch**

С помощью оператора switch программа может передать управление в одну из нескольких точек программы в зависимости от значения выражения.

switch (выражение) {

case *константа\_1:* оператор; break;

case *константа\_2:* оператор; break;

case *константа\_3:* оператор; break;

…

case *константа\_m:* оператор; break;

default: оператор;

}

*Выражение* должно иметь целочисленный тип. Значение выражения сопоставляется со всеми находящимися внутри switch-оператора case- константами.

*Оператор*, указанный после case-метки, выполняется, если значение switch-выражения равно соответствующей константе. Если ни с одной из case-констант совпадения нет, то управление передается на конструкцию с default-меткой, при условии ее наличия, в противном случае ни одна из подынструкций switch не выполняется.

Так как **case**-константы являются метками, то после найденного совпадения операторы будут выполняться последова­тельно до тех пор, пока не закончится **switch**-оператор. Поэтому после каждого блока операторов, относящегося к конкретной **case**-метке, необходимо указать ключевое слово (оператор) **break** (если это необходимо). Оператор **break** сразу же передает программное управление за пределы **switch**-оператора, завершая его.

**3.1. Оператор цикл for**

Цикл for является циклом с *предпроверкой* условия выхода. Конструкция **for**-цикла имеет синтаксис: for *(выражение1***;** *выражение2****;*** *выражение3) оператор;*

В *выражении1* производится ***инициализация*** переменных цикла. Если *выражение1* включает несколько операторов, то они должны быть разделены запятыми. *Выражение1 может* быть опущено в конструкции цикла, но точка с запятой после него должна быть оставлена. В качестве *выражения1* применяются присваивания или вызовы функций.

*Выражение2* формулирует ***условие выполнения цикла*** (т.е. если *выражение2* истинно, то операторы тела цикла выполняются, в противном случае происходит выход из цикла). Если *выражение2* опущено, то считается, что его значение всегда истинно. В последнем случае операторы тела цикла будут повторяться до тех пор, пока один из них не приведет к выходу из цикла.

*Выражение3* выполняется после каждого прогона цикла. Здесь обычно выполняются ***приращения*** переменных цикла. Если *выражение3* включает несколько операторов, то они должны быть разделены запятыми. *Выражением*3 может быть опущено в конструкции цикла. В *выражении3* обычно применяют присваивания или вызовы функций.

Раздел оператор может состоять из одного оператора или из блока операторов. Отдельные операторы должны всегда заканчиваться точкой с запятой. Если в качестве тела цикла выступает блок операторов, то он должен быть заключен в фигурные скобки.

**3.2. Бесконечный цикл for(;;)** представляет собой особый вид цикла for. Он получается, если все три элемента определения цикла for опустить и оставить только точки с запятой, разделяющие элементы.

Программно выход из такого цикла может быть осуществлен только с помощью оператора break , так как условие выхода из цикла не указано.

**3.4. Цикл do-while**

В языке С++ существует конструкция цикла do или do-while. Самое важное его отличие от цикла while состоит в том, что операторы тела цикла do выполняются хотя бы один раз. Оценка выражения, содержащего условие выхода из цикла, происходит всегда после выполнения тела цикла. Конструкция цикла имеет синтаксис:

do оператор

while (выражение);

**3.3. Цикл while**

Цикл while является циклом с предпроверкой условия выхода. Конструкция while-цикла имеет синтаксис: **while (выражение)**

**оператор; //**тело цикла

Если при вычислении выражения получается отличный от нуля результат, то выполняется оператор тела цикла, после чего выражение вновь вычисляется. Выражение должно иметь арифметический либо приводимый к нему тип.

В качестве *оператора* может выступать отдельный оператор или блок операторов. Отдельный оператор всегда должен за­канчиваться точкой с запятой (ограничителем операторов). Если тело цикла представляет собой блок операторов, то он должен быть заключен в фигурные скобки.

Выражение цикла while вычисляется прежде, чем выполняются операторы тела цикла.

Условием выхода из цикла является равенство нулю (false) выражения цикла.

Если при первом же вычислении выражения цикла while получено значение false (0), то операторы тела цикла не выполняются ни разу.

**4.1. Алгоритм Евклида**

Дана пара (a, b).  
Шаг 1. Поделить a на b с остатком r  
Шаг 2. Если r = 0, то НОД (a, b) = b  
Шаг 3. Если r <>0, то перейти к шагу 1 с парой (b, r)

do

if (a>b)

a%=b;

else

b%=a;

while ((a !=0) && (b!=0));

**4.2. Алгоритмы обмена чисел**

Имеется две переменные *a* и *b*. Поменять их значения местами.   
 int a,b, tmp;   
{1}   
 tmp = a; a = b; b = tmp;   
{2}  
 a = a + b; b = a - b; a = a - b;   
{3}   
a = a ^ b; b = a ^ b; a = a ^ b;

**4.3.А. Для заданного натурального числа определить количество единичных бит в его представлении.**

Суть алгоритма в том, что заводится битовая "маска", содержащая единственный ненулевой бит, позиция которого при работе цикла меняется от младшего к старшему.

При единичном значении очередного бита проверяемого числа, логическое умножение очередного значения маски на содержимое проверяемого числа маску не меняет.

int Count\_Bit(int a)

{

int bit;

int mask = 1;

int num = 0;

for ( int i = 0;i <=31;i++)

{

bit=mask & a;

if (bit == mask) num++;

mask = mask << 1;

}

return num;

}

**4.3.B. Найти количество повторений каждой цифры у заданного натурального числа**

void SumDigit (int a){

int b = a;

int c;

printf ("Povtorenie %d\n", a);

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

{

c = 0;

a = b;

while(a>0)

{

if(a%10 == i) c++;

a /=10;

}

if(c!=0)

printf ("%d => %d\n", i, c);

}

}

**4.3.C. Перевод из 10-й с/с в 2..9**

**int a,r;  
{ // ввод n  
a = 0; r = 1;   
 while (n >= 1)   
 {   
 a = a + (n % p) \* r; r = r \* 10;   
 n = n / p;   
}**

либо

int main()

{

char M1[16]={'0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F'};

int M[32]; //Массив под остатки

int a,i,n;

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

i=0;

while (a!=0)

{

M[i]=a%n;

a/=n;

i++;

}

for(i--;i>=0;i--)

printf("%c", M1[M[i]]);

printf("\n");

}

**D. Дихотомический поиск**

int BinSearch(int \*M, int left, int right, int x)

{

int mid;

while (left<=right)

{

mid=(left+right)/2;

if (M[mid]==x)

return(mid);

else

{

if (M[mid]>x)

right=mid-1; else

left=mid+1;

}

}

return(-1);

}

**E. Проверка, простое ли число**

int Prost (int a)

{ int i, n,flag;

double r;

if ((a==2)||(a==3))

flag =1;

else

{

i=2;

flag =1;

r=a;

n= (int)floor(sqrt (r));

while ((i <=n) && flag )

if (!(a%i)) flag=0;

else ++i;

}

return flag;

}

**F. Совершенные числа**

Натуральное число Р называется совершенным, если оно равно сумме всех своих делителей кроме Р. Доказано, что если р и 2р-1 - простые числа, то число Р=(2р-1)\*2^(p-1) является совершенным

void Number\_Sover (){

int n,k,p,i,flag,ch;

printf("Enter N ");

scanf("%d",&n);

p=2;i=1;

while (i<=n) {//простое или нет

int j=2;

flag=1; //да

while ((j<=p/2) && (flag))

if ( !(p%j)) flag=0;//нет

else ++j;

if ((flag)||(p==2) ||(p==3)){

int step=1; int m=p-1; int x=2;

while (m != 0){

if (m % 2 == 1) step=step\*x;

x=x\*x;

m=m / 2;

}

ch=step\*(step\*2-1);

printf("%d \n",ch);

i++;

}

p++;

}}

**E. Числа Мерсенна**

Числами Мерсенна называют числа вида 2^n-1

void Number\_Mersenne(int m)

{

printf("CM\n");

int r=0;

for (int i=1;i<=m;i++)

{

r=(2<<i-1)-1;

printf("%d\n",r);

}

}

**F. Числа Армстронга**

n-значное число называется числом Армстронга, если оно равно  
 сумме n-ых степеней своих цифр.

#include <stdio.h>

#include <math.h>

int main()

{

long a,b,k,c,chislo,z;

long double m,t,k1;

m=0; k=0;

printf("Input two borders a b (a<b): ");

scanf("%d %d", &a, &b);

for (int i=a; i<b+1; ++i){ //перебор всех чисел

m=0; k=0; z=i; chislo=i;

while (z) //определение количества знаков

{k++;

z/=10;}

k1=k;

for (int p=1; p<k+1; ++p) //подсчет суммы степеней цифр

{

c=chislo%10;

chislo=(chislo-c)/10;

t=pow(c,k1);

m+=t;

}

if (m==i)

printf("chislo Armstronga: %d\n", i);

}}

**G. Числа Фибоначчи**Число называется числом Фибоначчи, если оно является одним из членов последовательности:   
*f*n = *f*n-1 + *f*n-2,   
где *f*0 = 1 и *f*1 = 1.

Можно также определить n-й член ряда Фибоначчи, непосредственно подсчитав выражение:



**H. Числа Смита**

Число называется числом Смита, если сумма цифр числа равна сумме цифр разложения этого числа на простые множители.

**5.1. Объявление и определение функций .**

При объявлении функции необходимо указать типы параметров и имя функции. Имя функции – идентификатор, в скобках – формальные параметры.

Определение функции описывает, как она работает, т.е. какие действия надо выполнить, чтобы получить искомый результат.

int step(int ,int ); //объявление  
  
int step(int x,int n) //определение  
 {

//тело фнкции

return r;// возвращает результат вычислений и завершает выполнение функции.  
}

Функция может возвращать любые типы, кроме массива и функции.

Но функция может вернуть указатель на область памяти, в которой хранится массив и может вернуть указатель на функцию.

Если в качестве типа возвращаемого значения задан тип void, то не требуется возвращать значение.

**5.2. Функция main()**

main – это имя главной функции программы. С функции main всегда начинается выполнение. У функции есть имя (main), после имени в круглых скобках перечисляются аргументы или параметры функции (в данном случае у функции main аргументов нет). У функции может быть результат или возвращаемое значение.

**5.3. Функция main() с параметрами**

Конкретная реализация этой функции зависит от компилятора, но стандартом поддерживаются по меньшей мере два следующих формата вызова:

- Идентификатор\_типа main() { }  
- Идентификатор\_типа main  
 (int argc, char\* argv[], char \*envp[]) { }   
 параметры main - аргументы командной строки.

int main ( int argc , char\*\* argv , char\*\* envp);  
где   
*argc* – число параметров(слов) в командной строке, которые записываются в массив argv  
*argv* – массив указателей на строки  
*envp* – указатель на массив указателей на переменные среды

Последняя строка в массиве строк envp имеет нулевую длину.

**5.4. Обмен данными между функциями. Возвращаемое значение**

При вызове функции вместо формальных подставляются фактические параметры – значения, с которыми функция будет работать.

Для вызова функции используется конструкция

*имяфункции ( [ фактическиепараметры ] ),* т.е. вместо формальных параметров подставляются фактические параметры.

Фактический параметр может быть константой, переменной или более сложным выражением.

Независимо от типа фактического параметр он вначале вычисляется, а затем его величина передается функции.

Фактический параметр - это конкретное значение, которое присваивается переменной, называемой формальным параметром.

Есть функции, у которых отсутствует список формальных параметров, в этом случае записывают пустые скобки, также допускается указывать тип void.

При вызове функции необходимо указывать даже тогда, когда список фактических параметров пуст.

Функция может возвращать любые типы, кроме массива и функции.

Но функция может вернуть указатель на область памяти, в которой хранится массив и может вернуть указатель на функцию.

Если в качестве типа возвращаемого значения задан тип void, то не требуется возвращать значение.

**5.5. Параметры функции**

Все параметры в функции C передаются по значению, т. е. при вызове функции в стек заносится копия значения фактического параметра, и операторы в теле функции работают с этой копией. Поэтому значения фактических параметров, которые переданы в функцию, не изменяются. Передача параметров по значению предусматривает следующие действия:

1. При компиляции функции выделяются участки памяти для формальных параметров.   
2.Формальные параметры - это внутренние объекты функции. Для параметров типа float формируются объекты типа double. Для параметров типа char, short int создаются объекты типа int.   
3.Если параметром является массив, то формируется указатель на начало этого массива и он служит представлением массива-параметра в теле функции.

4. Вычисляются значения выражений, использованных в качестве фактических параметров при вызове функции.   
5. Значения выражений-фактических параметров заносятся в участки памяти, выделенные для формальных параметров функции.   
6. В теле функции выполняется обработка с использованием значений внутренних объектов-параметров, и результат передается в точку вызова функции как возвращаемое ею значение.   
7. Никакого влияния на фактические параметры функция не оказывает.

Для того чтобы обеспечить изменение передаваемых в функцию фактических параметров, необходимо явно передать адрес изменяемого параметра. Этого можно достичь двумя способами: 1) в качестве формального параметра описать указатель на тип, с которым будем работать; 2) использовать ссылки.

**5.6.A. Функция перевода числа в различные нумерации**

int System10( int n)  
{ int a = 0, r = 1;   
 while (n >= 1)   
 {   
 a = a + (n % p) \* r;

r = r \* 10;   
 n = n / p;   
}

**5.6.B. Перестановка цифр у целого числа (789->978)**

int Rev (int n)  
{ int tmp,digit, n\_dig;  
 tmp = n; n\_dig = 0;   
 while (tmp > 0)   
 {   
 tmp = tmp / 10;   
 n\_dig++;   
 }   
 tmp = n / 10;   
 digit = n % 10;   
 int res = digit \* Step(10, n\_dig - 1) + tmp;

//ф-я step пишется отдельно

return res;

}

**5.6.C. Поиск количества единичных бит в двоичном представлении числа.**

int Count\_Bit(int a)

{ int bit;

int mask = 1; int num = 0;

for ( int i = 0;i <=31;i++)

{

bit=mask & a;

if (bit == mask) num++;

mask = mask << 1;

}

return num;

}

**5.6.D. Поиск совершенных чисел с использованием функций**  
 int step(int x,int n)// функция xn см пример 7.3  
{int r=1;  
 while (n != 0)  
 { if (n % 2 == 1) r=r\*x;  
 x=x\*x;

n=n / 2;}  
 return r;

}

int Prime(int p)//функция проверки простое ли число  
{ int flag,d;   
 if ((p==2) ||(p==3))flag=1;//простое  
else{d=2;   
 flag=1; //простое  
 while ((d<=p/2) && (flag))  
 if ( !(p%d)) flag=0;//нет  
 else ++d; }  
 return flag;

}

void main()  
{ int p=2,i=1,n,ch; //написать ввод n  
 while (i<=n) // Вычисление N совершенных чисел  
 { if (Prime(p))  
 { ch=step(2,p-1)\*(step(2,p)-1);  
 cout <<ch<<endl;  
 i++;  
 }  
 p++;}}  
   
}

**5.6.E. Вычисление n!**

**long fact(int**);//объявление функции

void main()

{

int n;

printf("введите числa\n");

scanf("%d",&n);

printf("Факториал числа %d=%d\n",n,fact(n));

}

long fact(int num)  
 {   
 int res=1;  
 for (;num>1;num--)  
 res\*=num;  
 return res;  
}

**5.6.F. Вычисление сочетаний из N по M без факториала**

int Cnm(int n, int m)  
 {   
 int S=1;  
 for (int i=1; i<=m;i++)  
 S=S\*(n-i+1)/i;  
 return S;   
 }

**6. МАТЕМАТИЧЕСКИЕ ФУНКЦИИ.** В заголовочном файле <math.h> описываются стандартные математические функции и определяются макросы. Если аргумент выходит за область значений функции, то возникает ошибка области. Ошибка диапазона возникает тогда, когда результат функции не может быть представлен в виде double, происходит переполнение или потеря значимости. x и y имеют тип double, n - тип int, и все функции возвращают значения типа double. Углы в в радианах. **sin(x)** - синус *x,* **cos(x)** - косинус *x,* **tan(x)** - тангенс *x,* **asin(x)** - арксинус *x* в диапазоне *[-pi/2,pi/2]*, *x* в диапазоне *[-1,1],* **acos(x)** - арккосинус *x* в диапазоне *[0, pi]*, *x* в диапазоне *[-1,1];* **atan(x)** - арктангенс *x* в диапазоне *[-pi/2, pi/2]* , **atan2(y,x)** - арктангенс *y/x* в диапазоне *[-pi, pi]* , **sinh(x)** - гиперболический синус *x,* **cosh(x)** – гиперб. косинус x, **tanh(x)** - гиперболический тангенс *x,* **exp(x)** - Экспоненциальная функция *ex* , **log(x)** - натуральный логарифм *ln(x)*, *x > 0,* **log10(x)** - десятичный логарифм *lg(x)*, *x > 0,* **pow(x,y)** - *xy*, ошибка области, если *x = 0* или *y<=0* или *x<0* и *y* – не целое, **sqrt(x)** - квадратный корень *x*, *x >= 0.* **ceil(x)** – min целое в виде *double*, которое >= *x,* **floor(x)** - max целое в виде *double*, которое <= *x,* **fabs(x)** - абсолютное значение |x|, **frexp(x, int \*еxр)** - разбивает *x* на два сомножителя, первый из которых - нормализованная дробь в интервале [1/2,1), которая возвращается, а второй - степень двойки, эта степень запоминается в *\*exp*. Если *x* =0,то обе части результата =равны нулю; **modf(x,double \*ip)** - разбивается на целую и дробную части, обе имеют тот же знак, что и *x*. Целая часть запоминается в *\*ip*, дробная часть выдается как результат; **fmod(x, y)** - остаток от деления *x* на *y* в виде числа с плавающей точкой. Знак результата = знак *x*.

**7. ФУНКЦИИ ФОРМАТИРОВАННОГО ВЫВОДА**

Функция printf выводит аргументы, применяя к каждому определитель формата из \*format. Заголовок функции имеет следующий вид:  
**int printf(const char \*format [, arg, ...]);** format - указатель на строку знаков, содержащую два типа объектов: обычные знаки (отличные от %), которые выводятся неизмененными и спецификации преобразования, каждая из которых начинается с %.

Спецификации преобразования имеют следующую форму:   
**%[флаги] [ширина] [.точность] [модификатор] [тип]**  
флаги: любые символы, уточняющие формат вывода; ширина: минимальное число выводимых символов;точность: max после запятой;  
модификатор префикс: уточняет тип. Типы данных при выводе:  
**%** выводит знак процента (%), **%c** выводит символ, **%s** выводит знаки до достижения точности или NULL; считывает указатель на строку.   
**%d** выводит десятичное целое со знаком; считывает int (тоже само что i), **%i** выводит десятичное целое со знаком; (тоже само что d); **%o** выводит восьмеричное целое со знак; **%u** выводит десятичное целое без знака; **%x** выводит шестнадцатичное целое без знака (используя abcdef как цифры> 9); **%X** выводит шестнадцатичное целое без знака (используя ABCDEF как цифры > 9); **%f** выводит значение со знаком в виде [-]9999.9999; считывает число с плавающей точкой; **%e** выводит значение со знаком в виде [-]9.9999e[+|-]999; **%E** выводит тоже самое, что и при e, но использует E для записи экспоненты; **%g** выводит значение со знаком как в случае f или e, в зависимости от заданного значения и точности - нули на конце и десятичные точки печатаются только в случая необходимости; **%G** выводит тоже самое, что и при g, но использует E для записи экспоненты; считывает число с плавающей точкой; **%p** выводит указатель в формате данной реализации

Форматы вывода числовых данных со знаком.  
Для вывода целых чисел со знаком используется формат:  
 %[-] [+] [пробел] [0] [ширина] ] [.точность] [h|l] {d|i}  
-: выравнивание влево (по умолчанию - вправо)

пробел: выводит пробел в позицию знака  
ширина: минимальное число выводимых символов  
[.точность] : минимальное количество цифр, которые должны быть выведены  
h: модификатор short  
l: модификатор long  
d i: спецификация для вывода чисел

Формат вывода строки.   
%[-] [0] [ширина] [.точность] [h|l] [s]

Формат вывода действительных чисел.%[-] [#] [+|пробел] [0] [ширина] [.точность] [L|l] {f|e |E |g|G}

Форматы вывода числовых данных со знаком.  
Для вывода целых чисел со знаком используется формат:  
%[-] [+] [пробел] [0] [ширина] ] [.точность] [h|l] {d|i}  
-: выравнивание влево (по умолчанию - вправо)

пробел: выводит пробел в позицию знака  
ширина: минимальное число выводимых символов  
[.точность] : минимальное количество цифр, которые должны быть выведены  
h: модификатор short  
l: модификатор long  
d i: спецификация для вывода чисел

Формат вывода строки:  
%[-] [0] [ширина] [.точность] [h|l] [s]

Формат вывода действительных чисел.%[-] [#] [+|пробел] [0] [ширина] [.точность] [L|l] {f|e |E |g|G}

**8. ФУНКЦИИ ФОРМАТИРОВАННОГО ВВОДА**

Функция scanf вводит аргументы, применяя к каждому определитель формата из \*format. Заголовок функции имеет следующий вид:  
int scanf(const char \*format [, arg, ...]);  
Функция scanf вводит аргументы, применяя к каждому определитель формата из \*format.

Аргументы должны задавать адреса переменных. (&)  
Поведение scanf не опpеделено в слyчае нехватки аргументов для фоpматиpования.

scanf заканчивает работу, если встречает конец форматируемой строки. Если аргументов больше, чем требуется, то лишние аргументы игнорируются.

format - указатель на строку знаков, содержащую два типа объектов: спецификации преобразования, начинающиеся с % и управляющие символы.   
Спецификации преобразования имеют следующую форму:   
%[\*] [ширина] [модификатор] [тип]- ширина: мин число выводимых символов.  
 -модификатор префикс: уточняет тип.  
 - \* означает пропуск при вводе поля, определенного данной спецификацией.

**9. ФУНКЦИИ ОБЩЕГО НАЗНАЧЕНИЯ**

В заголовке <stdlib.h> объявляется набор функций, служащих для преобразования данных, генерации случайных чисел, получения и установки переменных среды, управления выполнением программ и выполнения команд.   
Abs - модуль целого числа   
bsearch - двоичный поиск   
qsort - сортировка массива

**Для получения псевдо-случайных чисел используют функции int rand(void);**

rand возвращает произвольные целые числа при каждом вызове; каждое число непредсказуемо выбирается алгоритмом, т.е. получаются случайные числа.

**void srand(unsigned int );**

Эту функцию можно применить для полyчения другой последовательности, использyя некотоpyю величинy.

Алгоритм зависит от статической переменной "random seed"; выдаваемые значения циклически повторяются через число вызовов rand, равное значению этой переменной.

**11.1. Объявление и инициализация массива**

тип<имя массива> [размер1][размер2]...[размер N];   
Количество элементов этого массива=размер1\*размер2\*...\*размер N;  
Одномерные массивы: тип<имя массива>[размер1];   
Двумерный массив: тип<имя массива>[размер1][размер2];

Имя массива - константный указатель.

Для одномерного массива:int days[12]={31,28,31,30,31,30,31,31,30,31,30,31};  
сhar Zifra[]={'0','1','2','3','4',’\0’};  
Для двумерного массива: int m[3][3]={{'1','2','3'},{'4','5','6'},{'7','8','\0'}};  
double temp[][3] = { { 3.2, 3.3, 3.4 }, { 4.1, 3.9, 3.9 } };

Вычислить размер пропущенной размерности  
const int size\_first = sizeof (temp) / sizeof (double[3]);   
 Функция sizeof определяет размер переменной или типа, т.е. это число = количеству байт памяти. Количество элементов в списке инициализации должно соответствовать размеру массива. Если список меньше размера массива, то элементы массива, на которых не хватило списка, будут забиты нулями. Если же список больше массива, то компилятор выдаст синтаксическую ошибку. Можно использовать пустые скобки для инициализации одномерных массивов (компилятор сам определит количество элементов в списке и выделит для массива нужное количество байт памяти). При инициализации многомерных массивов каждая размерность должна быть заключена в фигурные скобки. В языке С под массив всегда выделяется непрерывное место в оперативной памяти, количество байт памяти для одномерного массива: Количество байт = <размер базового типа> \* <количество элементов в массиве>. Можно не задавать размеры всех измерений массива, кроме самого последнего.

**11.2. Инициализация массивов случайными числами**#include <stdib.h>  
void main()  
{ int A[30],n;  
 **srand(1234);**  
 for (int i=0; i<n; i++)  
 { **A[i] = rand()%100;** printf("%d \n",A[i]);   
 }

}

**11.3. Ввод-вывод массивов**

Ввод элементов одномерного массива организовать цикл.   
 for (int i=0; i<n; i++)  
 {   
 printf ("введите элемент массива");  
 scanf("%d",&A[i]);  
 }

Вывод элементов одномерного массива.

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

printf("%d \n",A[i]);

// то же можно и через cout

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

printf("%d ",A[i]);

Вывод в 2 столбца:  
for ( i=0;i<n-1;i+=2)  
{  
 printf("%-5d%-5d\n",A[i],A[i+1]);  
 printf("%d\t%d\n",A[i],A[i+1]);  
}

Ввод элементов двумерного массива  
 for (int i=0;i<n;i++)  
 for (int j=0;j<m;j++)  
 scanf("%d",&Ma[i][j]);

// Вывод матрицы построчно, каждое число в 5 позициях  
for (int i=0;i<n;i++)  
 { for (int j=0;j<m;j++)  
 printf("%5d",Ma[i][j]);   
 printf(“\n"); }  
}

**11.4 Операции над элементами массива**

Над элементами массива можно выполнять любые операции, допустимые типом элементов (базовым типом): поиск, генерация(построение), преобразование, сортировки .

К двум совместимым статическим массивам А и В нельзя применять операция присваивания: т.к. это константный указатель.

int sum(int a[], int n)  
{

int s = 0;  
for (int i = 0; i < n; ++i)  
 s += a[i];   
return s;

}

int sum(int a[], int n)  
{   
for ( int i = -1,int s = 0;i < n-1; s += a[++i]);  
return s;  
}

**11.5. Передача массивов функциям**

Для передачи массивов-параметров функций формальный параметр массив можно объявить тремя способами:

-указатель, тип которого будет совпадать с типом элементов массива;  
 **int function (int \*a, int n) {……}**-массив с фиксированной длиной; **int function ( int a[20]) {……}** -безразмерный массив.  
 **int function ( int a[], int n) {……}**

Когда двумерный массив используется как параметр функции, необходимо передать указатель на первый элемент. Функция, получающая двумерный массив должна как минимум определять размерность первого измерения. Формальный параметр двумерный массив можно объявить следующими способами:   
**·        двумерный массив с фиксированной длиной;  
int function ( int ma[100][100]){……}** Для использования этой функции двумерный массив должен быть описан с максимальной размерностью int Ma[100][100], но можно обрабатывать размерности n\*m.

**·        двумерный с фиксированной размерностью первого измерения, т.е. второй размерностью; int func ( int ma[][100]) {…}** Для использования этой функции двумерный массив должен быть описан со второй размерностью =100

**·        указатель, которому при вызове буде соответствовать адрес первого элемента двумерного массива;**  
**int func ( int \*ma) {……}** В этом случае массив должен быть точно такой же размерности, как при вызове.

**·        указатель на двумерный массив, тип которого будет совпадать с типом элементов массива. int func ( int \*\*ma) {……}** - двумерный массив должен быть описан как указатель на указатель, для int \*\*ma необх.ВыделитьПамятьНекоторойРазмерности n\*m,но можно<= n\*m

**11.6. Обработка одномерных и двумерных массивов**

void main()  
{  
 int A[30],n;  
 srand(5000);  
 printf("Dimension? ");  
 scanf("%d",&n);  
 for (int i=0; i<n; i++)  
 {

A[i] = rand()%100;  
 printf("%d ",A[i]);  
 }  
 printf("\nsumma= %d ",sum(a, n));   
}

Поcтроение треугольника Паcкаля  
int k,l;  
 int Ma[10][10];  
 n=10;  
 Ma[0][0]=1;  
 for ( i=1; i<n; i++)  
 { Ma[i][i]=1;  
 Ma[i][0]=1;  
 for (int j=1; j<i; j++)  
 Ma[i][j]=Ma[i-1][j-1]+Ma[i-1][j];   
 }

**11.7. Поиск элемента с заданными характиристиками**

Поиск элемента в массиве  
 int Flag = -1;  
 int X=A[2];  
 for ( int i=1; i<n; i++)  
   
 if (A[i] = X)   
 { Flag = i;  
 break;

}

**12.1. Переменные-указатели**

Указатель – это переменная, которая содержит адрес памяти. Этот адрес как правило является адресом некоторой другой переменной.

Тип данных переменных, на который указывает указатель называется базовым типом данных. Стандартный вид объявления указателя:

Тип \* имя

int \*ip, \*iq; // указатели на целые объекты   
flоat \*fp; // указатель на символьный объект   
char \*cp; // указатель на символьный объект  
char \*const ср; // константный указатель на char  
char const\* pc; //указатель на константу типа char  
int x;

**12.2. Oператоры для работы с указателями.**

Унарные операторы \* и & имеют более высокий приоритет, чем арифметические операторы,

**Oператор &** - это унарный оператор, возвращает адрес объекта.

**Унарный оператор** **\*** возвращает значение переменной, находящейся по данному адресу. Для получения значение переменной х, адрес которой присвоен **ip**, нужно записать \*р. Этот оператор называют оператором ***разыменования или*** ***косвенного доступа***.

Можно выполнять преобразование типов указателей.

**float x=1., y, u;   
 float \*px;  
 int \*p;  
 px=&x;  
 p= (int\*)&x;  
 y=\*p;  
 u=\*px;**

**12.3. Арифметические действия с указателями.**

Над указателями можно выполнять следующие действия:

Вычитание указателей одного типа. В результате этой операции получается число типа int, которое указывает насколько один адрес смещен от другого в единицах, равных длине базового типа данных указателя.

К указателям можно применять операции **++, - -;** pm++; pn--;

К указателю можно применять суммирование или вычитание с целым числом.  
 pn=pn-2   
 pm=pm+4;

При обьявлении указателя желательно ему присвоить некоторое начальное значение. Если значение неизвестно, то присваивают NULL или 0**,** которое указывает, что указатель пока не используется.

**12.4. Динамическое распределение памяти**

Это можно выполнить двумя способами: с помощью функций языка C, и с помощью появившейся в C++ операции new.

Для выделения памяти в C используют следующие функции.  
**void \* malloc(size\_t size);**  
malloc выделяет в программе память размеров в size байт.  
int \*pi = (int \*) malloc(N \* sizeof(int));  
**void \*calloc(size\_t nitem, size\_t size);**calloc выделяет память размеров в (nitems\*size), при этом память обнуляется.

**void \*realloc(void\*ptr, size\_t size);**  
 realloc перераспределяет память, на который указывает указатель ptr, до размера size байт.

Для освобождения памяти используют функцию **void free(void\*ptr);**  
free освобождает память, на который указывает ptr. Прототипы этих функций находятся в <stdlib.h>.

Динамическое выделение памяти под массив из N элементов типа **int**  
**int \*pi = (int \*) malloc(N \* sizeof(int));   
int \*pi = new int[N];**

Освобождение памяти, занятой этим массивом, выполняется соответственно операторами  
**free(pi); delete [] pi;**

Двумерные массивы реализуются через указатели на указатели. Описание int a[10][10];  
соответствует описанию int (\*a)[10];

Двумерные массивы реализуются через указатели на указатели. Описание int a[10][10];  
соответствует описанию int (\*a)[10];

int \*\*a;

Динамическое выделение памяти под двумерный массив размерности M на N.  
1. Использование функции **malloc** требует записи следующего фрагмента программы:  
int \*\*mas;  
mas = (int \*\*) malloc(M \* sizeof(int \*));  
for (int i=0; i<N: i++)  
 mas[i] = (int \*) malloc(N \* sizeof(int));

Для корректного освобождения памяти необходимо записать следующий фрагмент:  
for (i=0; i<N: i++)  
 free(mas[i]);  
free(mas);

2. Использование операций **new** и **delete** приводит к аналогичным конструкциям:  
int \*\*mas = new int \*[M]  
 for (int i=0; i<N; i++)  
 mas[i] = new int [N];  
  
 for (int i=0; i<N; i++)  
 delete [] mas[i];  
delete [] mas;

**12.5. Указатели и массивы**

Имя массива - это константный указатель

Адрес массива можно присвоить обычному указателю  
int \*pa=a;

pa - адрес a[0],  
\*(pa+1) есть содержимое a[1]  
a+i - адрес a[i],   
\*(pa+i) - содержимое a[i].

Элемент массива можно изображать как в виде указателя со смещением, так и в виде имени массива с индексом.

Между именем массива и указателем, выступающим в роли имени массива, существует одно различие.

*Указатель - это переменная*, поэтому можно написать *pa=a* или *pa++*.

Имя массива *не является переменной*, и записи вроде *a=pa* или *a++* не допускаются.

**12.6. Массивы-параметры**

Для передачи массивов-параметров функций формальный параметр массив можно объявить тремя способами:

-указатель, тип которого будет совпадать с типом элементов массива;  
 **int function (int \*a, int n) {……}**-массив с фиксированной длиной; **int function ( int a[20]) {……}** -безразмерный массив.  
 **int function ( int a[], int n) {……}**

Когда двумерный массив используется как параметр функции, необходимо передать указатель на первый элемент. Функция, получающая двумерный массив должна как минимум определять размерность первого измерения. Формальный параметр двумерный массив можно объявить следующими способами:   
**·        двумерный массив с фиксированной длиной;  
int function ( int ma[100][100]){……}** Для использования этой функции двумерный массив должен быть описан с максимальной размерностью int Ma[100][100], но можно обрабатывать размерности n\*m.

**·        двумерный с фиксированной размерностью первого измерения, т.е. второй размерностью; int func ( int ma[][100]) {…}** Для использования этой функции двумерный массив должен быть описан со второй размерностью =100

**·        указатель, которому при вызове буде соответствовать адрес первого элемента двумерного массива;**  
**int func ( int \*ma) {……}** В этом случае массив должен быть точно такой же размерности, как при вызове.

**·        указатель на двумерный массив, тип которого будет совпадать с типом элементов массива. int func ( int \*\*ma) {……}** - двумерный массив должен быть описан как указатель на указатель, для int \*\*ma необх.ВыделитьПамятьНекоторойРазмерности n\*m,но можно<= n\*m размерности <= n\*m.

**12.7. Указатели на константы и константные указатели.**

Константный указатель - адрес в памяти, где хранится это число. Адрес в памяти постоянен, и такой указатель указывает, всегда на одну и ту же ячейку памяти, а содержимое этой ячейки может меняться.

Указатель на константу - адрес в памяти, где хранится элемент. Адрес, где хранится этот элемент, может изменяться, а содержимое этой ячейки - нет.

char\*p;  
chars[] = 'Ci++';  
const char\* pc = s; //указатель на константу  
  
pc[3] = 'g’;  
 // ошибка: рс указывает на константу  
  
рс = р; // правильно

char \*const cp = s; // констант-й указатель  
ср[3] = 'а, //правильно  
ср = р; //ошибка: ср - константа  
  
const char \*const cpc = s;  
// константный указатель на константу  
  
срс[3] = 'а'; ошибка: срс указывает на константу  
срс=р; //ошибка: срс является константой

**12.8. Указатели на функции.**

Имя функции является адресом этой функции в памяти.

Язык программирования Си позволяет определять указатели на функции. Например, следующая инструкция  
 int (\*fp)(int x, int y);

Пример  
int add(int x, int y)  
{ return x + y;}  
int sub(int x, int y)  
{ return x - y;}

void main()  
{  
 int (\*p)(int, int);  
 p = add;  
 /\* вызываем функцию через указатель \*/  
 printf("1 + 2 = %d\n", (\*p)(1, 2));  
  p = sub;  
 /\* можно вызвать функцию и так \*/  
 printf("1 - 2 = %d\n", p(1, 2));  
 return 0;

}

**12.9. Указатель на void**  
Существует указатель на объект любого типа**: void\*** имя; Действия с указателем:  
·        переменной типа void\* можно присвоить указатель на объект любого типа;

 · void\* нельзя разыменовывать.

 · void\* можно явно преобразовать в указатель на другой тип.

·  один void\* можно присвоить другому void\*;  
·  пару void\* можно сравнивать на равенство и неравенство;  
 · нельзя изменять void\* (размер объекта неизвестен)

Основными применениями void\* являются:  
 -передача указателей функциям, которым не позволено делать предположения о типе объектов  
 -возврат объектов «неуточненного типа» из функций.

Чтобы воспользоваться таким объектом, необходимо явно преобразовать тип указателя.

**12.9. Алгоритмы преобразование и перестановки .**

**void swaptmp(int \*px, int \*py) /\* перестановка \*px и \*py \*/  
{  
 // Обмен чисел   
 int temp;  
 temp = \*px;  
 \*px = \*py;  
 \*py = temp;  
}**

**void swap(int \*px, int \*py)   
*/\* перестановка \*px и \*py \*/*{  
 // Обмен чисел   
 \*px ^= \*py;  
 \*py ^= \*px;  
 \*px ^= \*py;  
}**

**int\* Revers(int \*a, int n)   
 // Переворачивание элементов массива   
{   
 int k;   
for (int i=0;i<n/2;i++)  
 {  
 k=a[i];  
 a[i]=a[n-i-1];  
 a[n-i-1]=k;   
 }   
 return a;}**

**int\* ReversP(int \*a,int n)  
{   
 int k;   
 for (int i=0;i<n/2;i++)   
 {   
 k=\*(a+i);   
 \*(a+i)=\*(a+n-i-1);  
 \*(a+n-i-1)=k;  
 }   
 return a;  
}**

**int\* ReversL(int \*a,int n)  
{   
 for (int i=0;i<n/2;i++)   
 {   
 \*(a+i)^=\*(a+n-i-1);   
 \*(a+n-i-1)^=\*(a+i);  
 \*(a+i)^=\*(a+n-i-1);  
 }   
 return a;  
}**

**12.10. Алгоритмы расширение и сжатие массивов**

**При обработке массивов можно вставлять и удалять элементы**void MoveRight(int \* a, int n, int num)  
{ //сдвигает все элементы на одну позицию вправо до номера num   
 for (int i=n; i>=(num+1); i--)   
 a[i]=a[i-1];   
}

void MoveLeft(int \* a,int \*n, int num)  
{ //сдвигает все элементы на одну позицию влево c номера num   
 for (int i=num; i<\*n-1; i++)   
 a[i]=a[i+1];   
 \*n=\*n-1;  
}

int\* DeleteCh(int \*a,int \*n, int item)

{

int i=0;

while (i<\*n)

{

if (a[i]==item)

{

MoveLeft(a, n, i);

//Сдвигаем и удаляем элемент a=(int\*)realloc(a,(\*n)\*sizeof(int));

i-- ;

}

i++;

}

return a;}

int\* InsertCh(int \*a,int \*n)

{ int i=0;

while (i<\*n)

{

if (a[i]%2==0)

{

a=(int \*)realloc(a,(\*n+1)\*sizeof(int));

MoveRight(a,(\*n)++,i+1); //Сдвигаем и добавляем элемент

a[i+1]=a[i];

i++ ;

}

i++;

}

return a;

}

**13.1Сортировка пузырьком**

Мы будем просматривать весь массив "снизу вверх" и менять стоящие рядом элементы в том случае, если "нижний" элемент меньше, чем "верхний". Таким образом, мы вытолкнем наверх самый "легкий" элемент всего массива. Теперь повторим всю операцию для оставшихся неотсортироваными N-1 элементов (т.е. для тех, которые лежат "ниже" первого. Как видно, алгоритм достаточно прост, но, как иногда замечают, он является непревзойденным в своей неэффективности.

void sort( int \*array, int size )

{

register int i, j;

int temp;

for( i = size - 1; i > 0; i-- )

{

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

{

if( array[j] > array[j+1] )

{

temp = array[j];

array[j] = array[j+1];

array[j+1] = temp;

}

}

}

return; }

**13.2Сортировка Вставками**

Будем разбирать алгоритм, рассматривая его действия на i-м шаге. Как говорилось выше, последовательность к этому моменту разделена на две части: готовую a[0]...a[i] и неупорядоченную a[i+1]...a[n].

На следующем, (i+1)-м каждом шаге алгоритма берем a[i+1] и вставляем на нужное место в готовую часть массива.

Поиск подходящего места для очередного элемента входной последовательности осуществляется путем последовательных сравнений с элементом, стоящим перед ним.

В зависимости от результата сравнения элемент либо остается на текущем месте(вставка завершена), либо они меняются местами и процесс повторяется.

void sort( int \*array, int size )

{

register int i, j;

int temp;

for( i = 1; i < size; i++ )

{

temp = array[i];

for ( j = i - 1; j >= 0; j--)

{

if( array[j] < temp )

break;

array[j+1] = array[j];

}

array[j+1] = temp; }}

**13.3Сортировка выбором**

На этот раз при просмотре мaccива мы будем искать наименьший элемент, сравнивая его с первым. Если такой элемент

найден, поменяем его местами с первым. В результате такой перестановки элемент с наименьшим значением ключа помещается

в первую позицию в таблице. Затем повторим эту операцию, но начнем не с первого элемента, а со второго. И будем продолжать

подобным образом, пока не рассортируем весь массив.

void sort( int \*array, int size )

{

register int i, j, k; int temp;

for( i = 0, k = 0; i < size - 1; i++, k++ )

{

temp = array[i];

for( j = i + 1; j < size; j++ )

{

if( temp > array[j] )

{

temp = array[j];

k = j;

}

}

temp = array[i];

array[i] = array[k];

array[k] = temp; }

return; }

**13.4Линейный поиск**

Линейный, последовательный поиск — нахождения заданного

элемента в некотором массиве. Поиск значения осуществляется простым сравнением очередного значения и искомого, если значения совпадают, то поиск считается завершённым. Если у массива длина = N, найти решение можно за N шагов. Сложность алгоритма - *O*(*n*). В связи с малой эффективностью линейный поиск используют только

если N не большое.

1. Определяем начало интервала поиска и выбираем элемент для поиска.

2. Cравниваем образец с выбранным элементом.

Если элемент совпадает с образцом, то определяем его индекс и переходим к шагу 4.

3. Если элемент не совпадает с образцом, то выбираем следующий, если он есть и переходим к шагу 2.

4. Если индекс определен, то элемент найден, иначе - искомого элемента в массиве нет.

int LineSearch (int \*a, int n, int x)

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

if (a[i] == x)

return i;

return -1; // элемент не найден

**Линейный выбор с подсчетом (сравнение и подсчет)**

Данный метод впервые упоминается в работе Э.Х. Фрэнда, хотя он и не заявил о ней как о своей собственной разработке.

При упорядочении таблицы этим методом необходима память для хранения исходной таблицы, а также дополнительно должна быть выделена память под счетчик для каждого элемента таблицы.

В этом алгоритме элементы не перемещаются. Он подобен сортировке таблицы адресов, поскольку таблица счетчиков определяет конечное положение элементов.

Просмотр таблицы начинается с первой записи. Ее ключ сравнивается с ключами последующих записей. При этом счетчик большего из сравниваемых ключей увеличивается на 1. При втором просмотре таблицы первый ключ уже не рассматривается, второй ключ сравнивается со всеми последующими. Результаты сравнений фиксируются в счетчиках. Для таблицы, содержащей N элементов, этот процесс повторяется N-1 раз.

После выполнения всех просмотров счетчик каждого элемента указывает, какое количество ключей таблицы меньше ключа этого элемента. Эти счетчики используются затем в качестве индексов элементов результирующей таблицы. Поместив записи в результирующую таблицу в соответствии со значениями их счетчиков, получим упорядоченную таблицу. Другими словами, если известно, что некоторый ключ превышает ровно 27 других, то после сортировки соответствующий элемент должен занять 28-е место.

/\* array - сортируемая таблица (массив) \*/

/\* count - таблица счетчиков (массив) \*/

/\* size - количество эллементов \*/

void sort( int \*array, int \*count, int size )

{

register int i, j;

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

count[i] = 0;

for( i = 0; i < size - 1; i++ )

{

for( j = i + 1; j < size; j++ )

{

if( array[i] < array[j] )

count[j]++;

else

count[i]++;

}

}

return;

}

**Сортировка Шелла**

Сортировка Шелла является довольно интересной модификацией алгоритма сортировки простыми вставками.

Рассмотрим следующий алгоритм сортировки массива a[0].. a[15].

1. Вначале сортируем простыми вставками каждые 8 групп из 2-х элементов (a[0], a[8[), (a[1],a[9]), ... , (a[7], a[15]).

2. Потом сортируем каждую из четырех групп по 4 элемента (a[0], a[4], a[8], a[12]), ..., (a[3], a[7], a[11], a[15]).

В нулевой группе будут элементы 4, 12, 13, 18, в первой - 3, 5, 8, 9 и т.п.

3. Далее сортируем 2 группы по 8 элементов, начиная с (a[0], a[2], a[4], a[6], a[8], a[10], a[12], a[14]).

4. В конце сортируем вставками все элементы

void Shella( int \*a,int n)

{

int h, i, t;

int k; // пpизнак пеpестановки

h=n / 2; // начальное значение интеpвала

while (h>0)

{ // цикл с yменьшением интеpвала до 1

// пyзыpьковая соpтиpовка с интеpвалом h

k=1;

while (k)

{

// цикл, пока есть пеpестановки

k=0;

for (i=0; i< n-h;i++)

{ //сpавнение эл-тов на интеpвале h }

if (a[i]>a[i+h])

{

t=a[i]; a[i]=a[i+h]; a[i+h]=t;

k=1; // пpизнак пеpестановки

}

}

}

h=h / 2; //{ yменьшение интеpвала } } }

**Бинарный поиск.**

Поиск элемента массива, ускоренного в сравнении споследовательным просмотром. Он известен как *метод дихотомии* или *метод половинного деления*. Как обычно, за скорость взимается плата: массив должен быть упорядочен. Сам по себе этап предварительного упорядочения, или *сортировки*, обходится недешево, во всяком случае – дороже однократного линейного поиска.

1.      Определяем середину интервала поиска.

2.      Сравниваем образец с элементом, расположенным посередине. Если образец оказался больше, то областью дальнейшего поиска становится правая половина; в противном случае - левая половина интервала, но в любом случае индексный интервал уменьшается вдвое. (Если осуществить еще одну проверку, то можно установить и совпадение, после чего дальнейшие шаги не обязательны.)

3.      Если остался интервал единичной длины, то переходим к заключительному шагу 4, в противном случае - к шагу 1.

4.      Либо единственный элемент интервала совпадает с образцом, либо - искомого элемента в массиве нет.

int BinarySearch(int a[],i, j, int x)

{

int , middle;

while (i <= j)

{

middle = (i + j)/2;

  if (x == a[middle])

return middle;

else

if (x > a[middle])

i = middle + 1;

else  
 j = middle - 1;

}

return -1;

}

**Сортировка Слияниями**

#include <stdio.h>

#include <stdlib.h>

void merge(int a[], int lb, int split, int ub)

{

int pos1=lb;

int pos2=split+1;

int pos3=0;

int \*temp =(int\*)malloc((ub-lb+1)\*sizeof(int));

while (pos1<=split && pos2<=ub)

{

if (a[pos1]<a[pos2])

temp[pos3++]=a[pos1++];

else

temp[pos3++]=a[pos2++];

}

while (pos2<=ub)

temp[pos3++]=a[pos2++];

while (pos1<=split)

temp[pos3++]=a[pos1++];

for (pos3=0; pos3<ub-lb+1; pos3++)

a[lb+pos3]=temp[pos3];

free(temp);

}

void mergsort(int a[], int lb, int ub)

{

int split;

if (lb<ub)

{

split = (lb + ub)/2;

mergsort(a, lb, split);

mergsort(a, split+1, ub);

merge(a, lb, split, ub);

}

}

void main()

{

int c[5];

for (int i=0;i<5; i++)

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

mergsort(c, 0, 4);

for (int i=0;i<5; i++)

printf("%d ", c[i]);

}

**Вызов стандартных функций сортировки и поиска.**

**Функция qsort** выполняют сортировку массива, элементы которого имеют произвольный тип. Эта функция реализует «быстрый алгоритм» сортировки массивов и имеет следующий прототип:

void qsort(void \*base, size\_t n, size\_t size, int (\*cmp) (const void \*e1, const void \*e2));

который описан в заголовочном файле stdlib.h. Кратко опишем назначение параметров этой функции:

base адрес массива,

n количество элементов в массиве,

size длина элемента массива,

cmp указатель на функцию сравнения, которая возвращает:

отрицательное число, если элемент e1 меньше элемента e2;

0, если элемент e1 равен элементу e2;

положительное число, если элемент e1 больше элемента e2.

**/\* функция для сравнения элементов массива \*/**

int comp\_int(const int\* e1, const int\* e2)

{

return (\*e1-\*e2);

}

**Функция bsearch** выполняет бинарный поиск элемента в отсортированном массиве. Эта функция имеет следующий прототип:

void\* bsearch(const void \*key, const void \*base,

size\_t n, size\_t size, int (\*cmp)(const void \*ck, const void \*ce);

который также описан в заголовочном файле stdlib.h. Первый параметр key этой функции является указателем на элемент, который нужно найти. Остальные параметры повторяют параметры функции qsort. В случае успешного завершения поиска функция bsearch возвращает адрес найденного элемента, а в случае неудачи NULL.

**/\* функция для сравнения элементов массива \*/**

int comp\_int(const int\* e1, const int\* e2)

{

return (\*e1-\*e2);

}

**Перевод чисел из одной системы счисления в другую.**

Программа переводит целое число из десятичной системы счисления в систему счисления с другим основанием(от двоичной до шестнадцатеричной)

с точностью до десяти знаков после запятой.

 #include <stdio.h>

#include <math.h>

void Revers(char \*a,int n)

{   
 int k;

// Переворачивание элементов массива

for (int i=0;i<n/2;i++)

{  
 k=a[i];

a[i]=a[n-i-1];

a[n-i-1]=k;

}

}

void Psystem(double re, int si ,char s[],int \*n)

{  
 int r,i,z=0;

double dre;

unsigned char c\_c[]="0123456789abcdef";

double y=modf(re,&dre);

int x=dre;

i=0;  
 while (x>=si)

{ r=x % si;

x=x / si;

s[i]=c\_c[r];

i++; }

s[i++]=c\_c[x];

Revers(s,i);  
 if (y!=0)

{ s[i++]='.';z=0;

do  
 { re=modf(re,&dre);

re = re\*si;

int j=re;

s[i++]=c\_c[j];

z=z+1;

}

while (re !=0 && z !=10);

} \*n=i;}

void main(void)

{

double re,dre;

int si,n;

char s[100];

printf("Введите основание системы (2..16) :");

scanf("%d",&si);

printf("Введите число: ");

scanf("%lf",&re);

Psystem(re,si ,s,&n);

PrintArray (s,n);

}

**14 В Си++ различают символьные и строковые константы и переменные.**

Символьная константа – это символ в одинарных кавычках. Она имеет тип char.

Строковая константа – это последовательность символов кода ASCII, заключённая: "...". Она имеет тип char[].

**Определение и инициализация строк.**

Строкой называется массив символов, который заканчивается символом '\0' (нулевой байт).

Поэтому строка объявляется как обычный символьный массив:

char \*str;

char str[10];

**Этот вид инициализации является краткой формой стандартной инициализации массива**

char str[]={‘T’,’g’,’\0’};

Без символа 0 мы имеем массив символов, а не строку.

Как и для других массивов, имя str является указателем на первый элемент массива:

str == &str[0], \*str == 'T', и \*(str+1) == str[1] == 'h'

**char heart[ ] = "Программирование на языке Cи++";**

**char \*head = "Программирование на языке Pascal";**

Основное отличие состоит в том, что указатель heart является константой, в то время как указатель head - переменной.

В том и в другом случае можно использовать операцию сложения с указателем.

**Указатели и строки**

Смысл заключается в том, что сама строка никогда не копируется. Оператор создает второй указатель, ссылающийся на ту же самую строку.

**Для ввода строки с консоли служит функция**

char\* gets(char \*str);

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.

**Сравнение строк**

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

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*.

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*.

**Копирование строк**

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.

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

*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*.

*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*

**Поиск символа в строке.**

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.

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.

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*.

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.

**Разбор на лексемы**

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.

**Память**

void \*memchr(const void \*str, int c, size\_t n);

Ищет первое вхождение с в n байтах str.

int memcmp(conct void \*str1, const void \*str2, size\_t n); Сравнивает первые n байт str1 и str2.

void \*memcpy(void \*str1, const void \*str2,size\_t n); Копирует n байт из str2 в str1

void \*memmove(void \*str1, const void \*str2,size\_t n); Копирует n байт из str2 в str1, перекрыв. Строк

*void \*memset(void \*str1, int c,size\_t n);*

Копирует c в n байт в str1.

**Функции преобразования строк и числовых данных.**

в заголовочном файле stdlib.h, который нужно включить в программу.

строки в число типа int

int atoi(const char \*str);

которая в случае успешного завершения возвращает целое число, в которое преобразована строка *str*, а в случае – неудачи 0. Например,

строки в число типа long

long int atol(const char \*str);

которая в случае успешного завершения возвращает целое число, в которое преобразована строка str, а в случае – неудачи 0. Например,

*строки в число типа double*

double atof(const char \*str);

которая в случае успешного завершения возвращает плавающее число типа double, в которое преобразована строка str, а в случае – неудачи 0.

**Функции преобразования строк и числовых данных.**

*long int strtol(const char \*str, char \*\*endptr, int base);*

преобразует строку *str* в число типа long int, которое и возвращает. Если аргумент *base* равен 0, то преобразование зависит от первых двух символов строки *str*:

если первый символ – цифра от 1 до 9, то предполагается, что число представлено в 10 c/c;

если первый символ – цифра 0, а второй – цифра от 1 до 7, то предполагается, что число представлено в 8 c/c;

если первый символ 0, а второй – ‘Х’ или ‘х’, то предполагается, что число представлено в 16 c/c.

Если аргумент *base* равен числу от 2 до 36, то это значение принимается за основание системы счисления и любой символ, выходящий за рамки этой системы, прекращает преобразование. В системах счисления с основанием от 11 до 36 для обозначения цифр используются символы от ‘A’ до ‘Z’ или от ‘a’ до ‘z’.

Значение аргумента *endptr* устанавливается функцией strtol. Это значение содержит указатель на символ, который остановил преобразование строки *str*. В случае успешного завершения функция strtol возвращает преобразованное число, а в случае неудачи – 0.

**Функции преобразования строк и числовых данных.**

unsigned long int strtol(const char \*str, char \*\*endptr, int base);

работает аналогично функции strtol, но преобразует символьное представление числа в число типа unsigned long int.

double strtod(const char \*str, char \*\*endptr);

преобразует символьное представление числа в число типа double.

Все функции, перечисленные в этом параграфе, прекращают свою работу при встрече первого символа, который не подходит под формат рассматриваемого числа.

Кроме того, в случае если символьное значение числа превосходит диапазон допустимых значений для соответствующего типа данных, то функции atof, strtol, strtoul, strtod устанавливают значение переменной *errno* в ERANGE. Переменная *errno* и константа ERANGE определены в заголовочном файле math.h. При этом функции atof и strtod возвращают значение HUGE\_VAL, функция strtol возвращает значение LONG\_MAX или LONG\_MIN, а функция strtoul – значение ULONG\_MAX.

Для преобразования числовых данных в символьные строки могут использоваться нестандартные функции itoa, ltoa, utoa, ecvt, fcvt и gcvt. Но лучше для этих целей использовать стандартную функцию sprintf

**int sprintf(char \*buffer, const char \*format, ...);**

форматирует строку в соответствии с форматом, который задан параметром *format* и записывает полученный результат в символьный массив *buffer*. Возвращает функция количество символов, записанных в символьный массив *buffer*, исключая завершающий нулевой байт.

%[флаги][ширина][.точность][модификаторы]тип

где

‘флаги’ – это различные символы, уточняющие формат вывода;

‘ширина’ определяет минимальное количество символов, выводимых по данной спецификации;

‘точность’ определяет максимальное число выводимых символов;

‘модификаторы’ уточняют тип аргументов;

‘тип’ определяет тип аргумента.

Для вывода целых чисел со знаком используется следующий формат вывода:

%[-][+|пробел][ширина][l]d

где

- – выравнивание влево, по умолчанию – вправо;

+ – выводится знак ‘+’, заметим, что для отрицательных чисел всегда выводится знак ‘-‘;

‘пробел’ – в позиции знака выводится пробел;

l – модификатор типа данных long;

d – тип данных int.

Для вывода целых чисел без знака используется следующий формат вывода:

%[-][#][ширина][l][u|o|x|X]

где

# – выводится начальный 0 для чисел в 8 c/c или начальные 0x или 0X для чисел в 16 c/c,

l – модификатор типа данных long;

u – целое число в 10c/c,

o – целое число в 8 c/c,

x, X – целое число в 16 c/c.

Для вывода чисел с плавающей точкой используется следующий формат вывода:

%[-] [+ | пробел][ширина][.точность] [f | e | E | g | G]

где

'точность' – обозначает число цифр после десятичной точки для форматов f, e и E или число значащих цифр для форматов g и G. Числа округляются отбрасыванием. По умолчанию принимается точность в шесть десятичных цифр;

f – число с фиксированной точкой,

e – число в экспоненциальной форме, экспонента обозначается буквой 'e',

E – число в экспоненциальной форме, экспонента обозначается буквой 'E',

g – наиболее короткий из форматов f или g,

G – наиболее короткий из форматов f или G.

**15 Цели введения новых типов:**

1. Ограничить множество значений существующего интервального типа.

2.   Определить тип, который описывает данные, имеющие сложную структуру.

3. Придать более осмысленные названия существующим типам данных

В первом случае используют перечисления - тип данных, который содержит необходимые элементы из базового типа.

Другие возможности для создания новых типов данных - объединения, перечисления, структуры.

Для переименования типов данных используется зарезервированное слово typedef.

Области видимости новых имен типов данных подчиняется правилам, что и имена переменных.

В первом случае используют перечисления - тип данных, который содержит необходимые элементы из базового типа.

Другие возможности для создания новых типов данных - объединения, перечисления, структуры.

Для переименования типов данных используется зарезервированное слово typedef.

Области видимости новых имен типов данных подчиняется правилам, что и имена переменных.

**Перечисления** - тип данных, который включает множество именованных целочисленных констант.

Переменные такого типа могут принимать значения только из заданного множества констант. Причем эти значения присваиваются по именам.

*Синтаксис:*  
enum имя\_типа {список имен}

Для хранения перечислений используются целочисленные значения.

Без явного указания нумерация начинается с 0 и увеличивается на 1. Можно указать другое значение,если его записать после знака = последующие значения будут

увеличиваться начиная с этого индекса

**enum** color {red, green, blue};

cout<<red<<' '<<blue<<' '<<green; *// "0 2 1"*

В языке С тип enum эквивалентен типу int.

Это позволяет присваивать целочислен-ные значения непосредственно переменным типа перечисления.

В C++ ведется более строгий контроль за типами данных, и такие присваивания не допускаются.

**color d=0**; // ошибка

**Структурой** называется упорядоченное множество данных. Данные входящие в структуру называются элементами (полями).

Тип каждого элемента может быть произвольным.

struct имя\_типа\_структуры

{  
список элементов;

// описание шаблона структуры

}

**struct emp**

**{  
int empno;**

**char value[10];**

**double salary;**

**};  
 struct emp a;** // определение структуры в языке C

emp a; // определение структуры в языке C++

Инициализируются также как и массивы.

**struct emp a={10,"Paul",200.15};**

Так как элементы структуры описываются в блоке, то их имена принадлежат локальной области видимости (внутри этого блока).

Для доступа к элементу используется оператор (**.**).

**a.empno=20;  
strcpy(a.value,"John");  
a.salaty=3000;**Для структур одного типа можно использовать оператор присваивания, который копирует поля структуры.

**struct emp b;**

**b=a;**

**Доступ к отдельным элементам структуры**

Для доступа к отдельным полям структурной переменной используют оператор «.» (точка), формируя ссылку на нужное поле из имени структурной переменной и

имени поля.

*имя\_структурной\_переменной . имя\_поля*

 Для доступа к полям структурной переменной через указатель используется оператор «->» (минус больше)

*указатель\_на\_структуру -> имя\_поля*

***Передача структур в функции.***

*Когда структура используется как параметр функции она передается по значению.*

*//печать ;*

*void print(struct person reader)*

*{****printf (" %s, %d, %s\n",***

***reader. name, reader.post\_code,reader.region);***

*}*

Если необходимо, чтобы функция изменяла значения полей структуры, то как параметр в функцию передается указатель на структуру.

Для доступа к элементу используется оператор (->).

***Объединения*** *(****union****)****.***

Объединением называется область памяти, использующаяся для хранения данных разных типов.

Причем одновременно в объединении могут храниться данные только одного определенного типа.

Данные входящие в объединение называются его полями (элементами). Тип описывающий объединение также называется объединением.

С помощью объединения к одной и той же области памяти можно обращаться как к данным различного типа.

Объединение можно рассматривать как структуру, все элементы которой имеют одинаковый начальный адрес в памяти.

Длина объединения определяется наибольшей из длин полей объединения. Объединения позволяют сэкономить память в том случае, если необходимо

хранить только один из элементов объединения, но заранее неизвестно, какой именно.

union *[имя\_типа объединения] {*

*тип\_1 элемент\_1;*

*тип\_2 элемент\_2;*

*…*   
*тип\_n элемент\_n;* *}*

union num

{  
int n;

double f;

}

union num d; //d-переменная типа num (язык C)

num d; // (C++)

Имена элементов переменной типа объединения принадлежат локальной области видимости внутри блока. **Для доступа к элементам объединения используется оператор (.)**

num e

e.n=1;

e.f=2.7;

Над объединениями одного типа возможна операция присваивания G=e;

**Для доступа к элементам объединения на которые указывает указатель используется оператор(->)**

Объединение (как и структура) не может содержать в качестве элементов переменные собственного типа.Но в качестве элементов объединения могут использоваться указатели или ссылки на переменные собственного типа объединения

**Битовые поля**

**Поля битов** – это особый тип полей структуры. Они используются, если нужно запомнить данные, для которых необходима ячейка памяти менее одного байта. Это

могут быть, например, флажки, которым достаточно одного бита для представления значений в двоичной логике. Битовыми полями называются элементы структуры или объединения, которые определяется как последовательность битов и описывается следующим образом.

**Синтаксис описания битового поля следующий:**

***Идентификатор\_типа [имя\_поля]: ширина\_поля*;**

**struct status {**

**unsigned Shadow : 1;**

**unsigned Border : 1;**

**}**

К битовым полям не может быть применена операция получения адреса (**&**) и поэтому не существует указателей на поля. Непоименованные битовые поля с шириной поля 0 задают выравнивание следующего поля по границе слова.

**Инициализируются битовые поля как обычные элементы структуры.**

**status a={0,1,1,2,1};**

**16 Файл** – последовательность байтов, записанных на внешнем носителе информации.

**Поток** – логический интерфейс (программа) который обеспечивает доступ программы-пользователя к файлу. Прежде чем использовать поток для доступа к файлу, его необходимо соединить с файлом. Эта информация хранится в структуре типа FILE, поэтому поток имеет вид FILE\* (их часто отождествляют). Когда поток соединяют с файлом, то говорят, что файл открывают; когда отсоединяют - говорят, что файл закрывают. Каждый поток может работать в двух режимах: текстовом и бинарном. Режим работы потока задается при его соединении с файлом. В текстовом режиме поток записывает и читает из файла текстовые строки, которые заканчиваются символом ‘\n’ и могут содержать символ ‘\t’. По стандарту поток должен обеспечивать обработку строк длиной не менее 254 символа, включая символ ‘\n’.

В бинарном режим поток записывает и читает данные из файла в том виде, в котором они хранятся в оперативной памяти.

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

которая открывает файл, имя которого задано параметром *filename*, в режиме, заданном параметром *mode*. В случае успешного завершения функция fopen возвращает указатель на поток, а в случае неудачи – NULL.

Параметр *mode* может принимать следующие значения:

“r” – чтение в текстовом режиме,

“w” – запись в текстовом режиме,

“a” – присоединение в текстовом режиме,

“rb” – чтение в бинарном режиме,

“wb” – запись в бинарном режиме,

“ab” – присоединение в бинарном режиме,

“r+” или “w+” или “a+” – чтение и запись в текстовом режиме,

“r+b” или “w+b” или “a+b” – чтение и запись в бинарном режиме,

“rb+” или “wb+” или “ab+” – чтение и запись в бинарном режиме.

**int fclose(FILE\* stream);**

которая закрывает файл, при этом освобождая все буферы потока. При успешном завершении функция возвращает 0, а в случае неудачи – EOF.

Для перенаправления потока используется функция:

**FILE\* freopen(const char\* filename, const char\* mode, FILE\* stream);**

которая закрывает файл, соединенный с потоком *stream*, и соединяет с этим потоком файл *filename* в режиме *mode*. В случае успеха функция возвращает указатель на поток, а в случае неудачи – NULL. Параметр *mode* принимает те же значения, что и в функции fopen

**Работа с индикатором ошибки**

С каждым потоком связан индикатор ошибки, который находится в установленном положении, если в потоке, связанном с файлом произошла ошибка. В противном случае индикатор ошибки находится в сброшенном состоянии. Для работы с индикатором ошибки используются функции ferror и clearerr.

int ferror(FILE\* stream);

возвращает ненулевое значение, если индикатор ошибки установлен, в противном случае – возвращает 0.

void clearerr(FILE\* strem);

сбрасывает индикаторы ошибки и конца файла для потока *stream*.

Структура FILE содержит индикатор конца файла, который устанавливается в ненулевое значение функцией чтения из файла при достижении этой функцией конца файла. Состояние конца файла читается функцией

**Конец файла**

**int feof(FILE\* stream);**

которая возвращает ненулевое значение, если индикатор конца файла установлен, в противном случае функция возвращает 0.

**Работа с индикатором позиции.**

void rewind(FILE\* stream);

устанавливает индикатор позиции на начало файла, связанного с потоком *stream*. При этом сбрасывается индикатор ошибки и конца файла.

*int fseek(FILE\* stream, long offset, int mode);*

сдвигает индикатор позиции файла на *offset* байтов. В случае успешного завершения функция возвращает 0, в противном случае – ненулевое значение. Параметр *mode* указывает на режим сдвига и может принимать следующие значения:

SEEK\_SET – смещение от начала файла,

SEEK\_CUR – смещение от текущей позиции,

SEEK\_END – смещение от конца файла.

При работе с текстовым потоком должны использоваться только следующие комбинации значений параметров:

*mode* = SEEK\_SET offset = 0 или

offset = значению, возвращаемому функцией ftell

*mode* = SEEK\_CUR offset = 0

*mode* = SEEK\_END offset = 0

*int fsetpos(FILE\* stream, const fops\_t \*pos);*

устанавливает индикатор позиции файла *stream* в позицию, на которую указывает параметр *pos*. Индикатор конца файла сбрасывается. В случае успеха функция возвращает 0, а в случае неудачи возвращает ненулевое значение и устанавливает переменную *errno*.

*long ftell(FILE\* stream);*

в случае успешного завершения возвращает текущую позицию файла *stream*, а в случае неудачи – возвращает значение 1L и устанавливает значение переменной *errno*. Для бинарного потока позиция равна смещению в байтах от начала файла, а в случае текстового потока – значению, которое может использоваться функцией fseek.

*int fgetpos(FILE\* stream, fops\_t\* pos);*

записывает текущую позицию файла *stream* по адресу *pos*. В случае успеха функция возвращает 0, а в случае неудачи – ненулевое значение и устанавливает значение переменной *errno*.

**Блочный ввод-вывод**

*size\_t fwrite(const void\* ptr, size\_t size, size\_t nitems, FILE\* stream);*

которая записывает содержимое блока памяти, на который указывает ptr, в файл stream. Длина записываемого блока определяется как произведение size\*nitems. Функция возвращает число записанных единиц памяти. В случае удачи это число должно быть равно nitems.

*size\_t fread(const void\* ptr, size\_t size, size\_t nitems, FILE\* stream);*

Для чтения блока из файла используется функция параметры которой имеют тот же смысл, что и в функции fwrite

#### **Удаление и переименование**

#### **int remove(const char\* filename);**

#### Удаляет файл с именем filename. В случае удачи возвращает 0, иначе возвращает не 0.

#### **int rename(const char\* old\_filename, const char\* new\_filename);**

#### Переименовывает файл. В случае удачи возвращает 0, иначе возвращает не 0.

**Символьный ввод-вывод** используется с текстовыми потоками.

*int fputc(int c, FILE\* stream);*

записывает символ, заданный параметром *с*, в поток *stream* и продвигает индикатор позиции на следующий символ. В случае успеха функция возвращает символ *с*, а в случае неудачи EOF и устанавливает индикатор ошибки.

*int putc(int c, FILE\* stream);*

#### работает так же как и функция fputc, но может быть реализована как макрокоманда.

#### *int fgetc(FILE\* stream);*

#### читает символ из потока *stream* и продвигает индикатор позиции на следующий символ. В случае успешного завершения функция возвращает прочитанный символ. В случае достижения конца файла функция возвращает EOF и устанавливает индикатор конца файла. В случае ошибки функция возвращает EOF и устанавливает индикатор ошибки.

#### *int getc(FILE\* stream);*

#### работает так же как и функция fgetc, но может быть реализована как макрокоманда.

#### 

#### *int ungetc (int c, FILE\* stream);*

#### записывает символ *с* в поток *stream*. Функции fseek, fsetpos и rewind игнорируют такие символы. Доступ к записанным символам выполняется по правилу FIFO. В случае успеха функция возвращает записанный символ, а в случае неудачи – EOF.

#### **Форматированный ввод-вывод.**

#### *int fprintf(FILE\* stream, const char\* format, …);*

#### выполняет вывод в файл *stream* в соответствии с форматной строкой *format*.

#### 

#### *int fscanf(FILE\* stream, const char\* format, …);*

#### выполняет ввод из файла *stream* текста в соответствии с форматной строкой *format*.

#### **Стандартные потоки.**

#### Каждой программе предоставляются три стандартных потока, которые по умолчанию соединены с консолью. Указатели на эти потоки возвращают макрокоманды stdin, stdout, stderr. Для работы со стандартными потоками предназначены рассмотренные ранее функции scanf, printf, а также следующие функции:

#### int putchar(int c); /\* вывод символа в stdout \*/

#### int getchar(void); /\* ввод символа из stdin \*/

#### int puts(const char\* str); /\* вывод строки в stdout \*/

#### char\* gets(char\* str); /\* ввод строки из stdin \*/

#### void perror(const char\* str); /\* вывод сообщения об ошибке в stderr \*/

#### Работают эти функции так же, как и аналогичные функции для работы с файлами

#### Использование: выводим строку str в стандартный поток stderr, после этого выводим пробел и сообщаем об ошибке, связанной со значением переменной errno.

**17 Абстрактный тип данных “список”.**  
Списки - это структуры данных, которые доступны для вставки и удаления элементов в любой позиции списка.

**Односвязные списки -** это списки перемещение по которым возможно только в прямом или в обратном направлении.

**Двухсвязные** **списки -** это списки перемещение по которым возможно как в прямом так в обратном направлении

**Реализация списков с помощью указателей.**struct node /\* элемент списка \*/  
{  
int item; // данные   
node\* next; //указатель на следующий элемент   
node\* prev; //указатель на предыдущий элемент   
};

node \*head, \*tail; // указатели на первый и последний элементы в списке   
 **node \*head, \*tail;**

**Вставка элементa в начало списка**  
1. Создать новый элемент типа список  
2. Инициализировать его информационное поле.  
3. Следующей его ссылке присвоить адрес первого элемента (вершины списка).  
4. Предыдущей ссылке присвоить значение NULL.  
5. Проверить, не пустой ли список   
6. Если пустой, создать список из одного элемента с пустыми ссылками  
7. Изменить адрес вершины списка.

void push\_front(int n)   
{  
 node\* tmp = new node;  
 tmp->item = n;  
 tmp->next = head;   
 tmp->prev = 0;   
 if(head)  
 head-> prev = tmp;  
 else  
 tail = tmp;  
 head = tmp;  
}

**Вставка элементов в конец списка**1. Создать новый элемент типа список  
2. Проверить, не пустой ли список   
3. Инициализировать его информационное поле.  
4. Следующую его ссылку сделать пустой  
5. Предыдущей присвоить адрес последнего элемента.  
6. Указателю последнего присвоить адрес нового элемента.   
7. Если пустой, создать список из одного элемента с пустыми ссылками

void push\_back(int n)   
{  
 node\* tmp = new node;  
 tmp->item = n;  
 tmp->next = 0;  
 tmp->prev = tail;  
 if(tail)  
 tail->next = tmp;  
 else  
 head = tmp;  
 tail = tmp;  
}

**Добавление элементa в середину.**

**Вставляем элемент в позицию pos+1**1. Создать новый элемент типа список  
2. Инициализировать его информационное поле.  
3. Если пустой, создать список из одного элемента с пустыми ссылками.  
4. Найти позицию для вставки.  
5. Следующей ссылке нового элемента присвоить адрес следующего элемента .  
6. Предыдущей ссылке - адрес найденного.  
7. У предыдущего элемента следующей ссылкой сделать адрес созданного элемента.   
8. У следующего элемента предыдущей ссылкой сделать адрес созданного элемента

}

void insert(int data, int pos)   
/\* вставить в позицию pos+1 \*/  
{ node\* tmp = new node;  
 tmp->item = data;  
 node\* ptmp = head;  
 if(ptmp == 0)  
 { tmp->next = 0;  
 tmp->prev = 0;  
 head = tmp;  
 tail = tmp;  
 return;

int i = 0;  
 while ((i<pos)&&(ptmp->next))  
 { ptmp = ptmp->next;  
 i++;  
 }  
if (ptmp->next == 0)  
 { ptmp->next = tmp;  
 tmp->prev = ptmp;  
 tmp->next = 0;  
 tail = tmp;  
 return;  
 }

ptmp->next->prev = tmp;  
 tmp->next = ptmp->next;  
 ptmp->next = tmp;  
 tmp->prev = ptmp;}

**Удалить элемент c позиции pos.**1. Найти адрес элемента для удаления.  
2. У предыдущего элемента следующей ссылкой сделать следующую ссылку найденного элемента.   
3. У следующего элемента предыдущей ссылкой сделать предыдущую ссылку найденного элемента  
4.Удалить найденный

int remove (int pos)  
 { node\* tmp = head;  
 for ( int i = 0; i<pos; i++)  
 { if(tmp == 0) return 0;  
 tmp = tmp ->prev; }  
 tmp->next->prev = tmp->prev;  
 if(tmp ->prev != 0)  
 tmp->prev->next = tmp->next;  
 int res = tmp->item;  
 delete tmp;  
 return res; }

Исключить элемент из начала списка int pop\_front()   
{ if (head)  
 { int res = head->item;  
 node \*tmp = head;  
 if ( head->next)  
 head->next->prev = 0;  
 else tail = 0;  
 head = head->next;  
 delete tmp; return res;}  
else return 0; }

Исключить элемент из конца списка   
 int pop\_back()  
{ if(tail)  
 { int res = tail->item;  
 node \*tmp = tail;  
 if(tail->prev) tail->prev -> next = 0;  
 else head = 0;  
 tail = tail->prev;  
 delete tmp;  
 return res;  
 }return 0;}

**Стек -** это специальный тип списка, в котором все вставки и удаления выполняются только на одном конце, называемом вершиной (TOP или HEAD). Для обозначения стеков используют аббревиатуру LIFO (Last-in-First-out)

Для обработки стека обычно используют следующие методы: **Init** Создает пустой стек **PEEK** Возвращает элемент из вершины стека **POP** Удаляет элемент из вершины стека **PUSH** Вставляет элемент в вершину стека **EMPTY** Возвращает True, если стек пустой

struct stack  
{

int item;  
stack \*prev;

};  
stack \*top;  
void InitStack ()   
{

top = NULL;  
}

void push (int a)   
**//Занести в стек**  
{  
 stack \*temp;  
 temp = new stack;  
 temp->prev = top;  
 temp->item = a;  
 top = temp;  
}

int pop ( )   
**//Извлечь из стека**  
{  
 int res;  
 stack \*temp;  
 temp = top;  
 top = top->prev;  
 res = temp->item;  
 delete temp;  
 return res;  
}

int peek ()  
**// Выбор элемента из верхушки стека**{  
 if (top) return top->item;  
}

bool is\_empty ()

**//Проверка пуст ли стек**  
{ bool res = false;  
if (top == NULL)  
res = true;  
return res;

}

void removestack ()  
**//Удаление содержимого стека**  
{  
 stack \*temp;  
 temp = top;  
 while (top)  
 {  
 temp = top;  
 top=top->prev;  
 delete temp;

}

}

**СТЕК с использованием массива**  
int Size = 100 ;  
 int size; /\* размерность массива \*/  
 int\* p; \* указатель на массив \*/  
 int top; /\* верхушка стека \*/  
 int\* ArrayStack(int \_size)   
{   
 size = \_size > Size ?\_size : Size;  
 p = new int[size];  
 top = -1;  
 return p;   
}

**bool isEmpty() /\* пустой стек? \*/  
{  
 return top == -1;  
}  
  
bool isFull() /\* полный стек? \*/  
{  
 return top + 1 == size;  
}**

void DeleteStack()   
 **/\* Удаление стека \*/**  
{  
 top = -1;  
 size = 0;  
 delete []p;  
 p = 0;  
}

int push(const int n)   
**/\* втолкнуть элемент в стек \*/**  
{  
 if (isFull ( ) )  
 return -1;  
 top++;  
 p[top] = n;  
 return top;  
}

int pop ( )   
 **/\* вытолкнуть элемент из стека \*/**  
{ if(!isEmpty())  
 { int res = p[top];  
 top--;  
 return res;}  
 return 0;  
}

**Преобразовать инфиксную строку-выражение в постфиксную строку-выражение и вычислить ее результат**.

Инфиксная  
 (A+B\*(D-E))/(F+G)  
Префиксная

/\*+AB-DE+FG

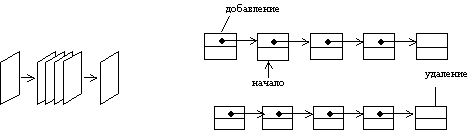
Постфиксная

ABDE-\*+FG+/

Алгоритм:  
 1. Числа(буквы) записываются в строку том же порядке, в каком встречаются в исходном выражении.  
2. Найденный символ арифметической операции заносится в стек при условии, что его приоритет выше приоритета предыдущего символа операции. В противном случае в строку записывается предыдущий символ операции, а найденный заносится в стек.

3. Открывающая скобка заносится в стек. Считается, что ее приоритет ниже приоритета всех арифметических операций. При нахождении закрывающей скобки все содержимое стека до первой открывающей скобки удаляется и записывается в строку. Открывающая скобка удаляется из стека и в строку не записывается.

**Очередь** - это специальный тип списка, в котором все вставки выполняются в одном конце, называемом последним (REAR),  
а удаления выполняются в другом конце, называемом передним (FRONT).Для обозначения стеков используют аббревиатуру FIFO (First-in-First-out: первым вошел – первым вышел).



***Включение элемента в очередь***  
1. создать новый элемент типа очередь  
2. инициализировать его информационное поле.  
3. изменить его указатель  
4. изменить указатель на последний элемент очереди так, чтобы последним стал новый элемент.

***Выбор элемента из очереди***  
1. Информационное поле первого ее элемента присвоить результирующей переменной,  
2. Элемент исключить из очереди и удалить.  
3. Может быть проверка на то, являлся ли этот элемент в очереди единственным, и если да, то необходимо соответствующим образом изменить указатель на последний элемент

**18. ФУНКЦИИ И РАЗРАБОТКА ПРОГРАММ**

**18.1. Функции с переменным количеством параметров**

В языке C допускаются функции с переменным количеством параметров. В этом случае в конце списка параметров ставится “…”  
 При этом предполагается, что явно указан хотя бы один параметр.  
int print (int n, char \* str , …);

Для обработки функции с переменным количеством параметров используется переменная типа va\_list и три макрокоманды из библиотеки stdarg.h:  
va\_start ( va\_list ap , last\_arg);  
 Инициализирует параметр ap , где last\_arg – последний известный фиксированный элемент функции.

**type va\_arg (va\_list ap, type);**  
 Возвращает следующий аргумент типа type, **ap** должен быть предварительно инициализирован макрокомандой **va\_start;**   
**void va\_end (va\_last ap);**  
 Обеспечивает правильную работу инструкции return, после вызова va\_end работа макрокоманды va\_arg невозможна.

**18.2. Глобальные и локальные переменные, область видимости и время существования.**

Если переменная объявлена вне какой-либо функции, то она называется глобальной. Если переменная объявлена внутри функции, то она называется локальной

Область видимости и время существования переменной определяются спецификаторами классов памяти.

Aвтоматические переменные auto существуют - временно, область действия - локальная;

Глобальная переменная с определенным именем может быть определена единожды, продолжительность существования внешней переменной постоянная, область действия глобальная (все файлы)

Переменные типа static имеют такую же область действия, как автоматические переменные, но они не исчезают, когда содержащая их функция закончит свою работу. Компилятор хранит их значения от одного вызова функции до другого.

Регистровые переменные запоминаются в регистрах центрального процессора. Спецификатор static можно применять только в локальных переменных и параметрах функций.

**18.3. Спецификаторы классов памяти.**

Спецификаторы классов памяти указывают как долго должна храниться объявленная переменная и определяют область видимости этой переменной.   
Существуют 4 спецификатора классов памяти:   
·  auto · extern · static   ·  register.

Автоматические переменные  
**аuto –** указывает, что переменная, определенная внутри функции, является локальной. Этот спецификатор используется по умолчанию. Aвтоматические переменные существуют - временно, область действия - локальная; Переменная, описанная вне функции, является внешней.  
**Extern –** указывает, что переменная является глобальной и определена в другом файле программы. Глобальная переменная с определенным именем может быть определена единожды, продолжительность существования внешней переменной постоянная, область действия глобальная (все файлы)

**Static –** для глобальной переменной ограничивает область её видимости только файлом, в котором она определена; для локальной – сохраняет её и значение до конца работы, это означает, что такие переменные остаются в работе.

**Register –** указывает, что значение переменной должно храниться в регистрах процессора. Этот спецификатор можно применять только в локальных переменных и параметрах функций.Регистровые переменные запоминаются в регистрах центрального процессора, где доступ к ним и работа с ними выполняются гораздо быстрее, чем в памяти.

**18.4. Компоновка нескольких файлов в одну программу**

С помощью компилятора текст преобразуется в исполняемый файл. Обработка исходных файлов происходит в три этапа. Сначала файл обрабатывается препроцессором, который выполняет операторы **#include, #define** и еще несколько других.   
 Затем, на **втором этапе,** компилятор создает **объектный файл**, в котором имеются ссылки на различные системные функции и на стандартные функции языка C++.  
 На **третьем этапе** компиляции к объектному файлу подсоединяются все функции на которые он ссылается.

Основная цель многоэтапной компиляции программ – возможность компоновать программу из многих файлов. Каждый файл представляет собой законченный фрагмент программы, который может ссылаться на функции, переменные или классы, определенные в других файлах.

**Исполняемая программа представляет собой непрерывный блок, который состоит из четырех частей**:  
--**Програмный код  
--Глобальные переменные   
--Куча (Heap) , где хранятся динамические переменные  
--Стек (stack).**

**В стеке хранятся:**

-Адреса возврата из функций  
-Значения аргументов функций  
-Локальные переменные

В языке Си существует строгое правило, в соответствии с которым все переменные и функции должны быть объявлены или определены.

В программе должно быть только одно определение функции. Удобно было бы поместить Определение функции в отдельный файл, а в других файлах в начале помещать лишь объявление, прототип функции.

**Пример использования указателей на функции**  
Последовательность действий следующая:  
**Создать файл point\_function.cpp**

**Создать файл function.cpp**

**Создать файл function.h**

**18.5. Директивы препроцессора**

Назначение препроцессора - обработка исходного текста программы до ее компиляции.

Препроцессор обрабатывает собственные *директивы*. Эти директивы задаются, как правило, в отдельной строке и начинаются с символа ’#’.

В языке C++ можно задать следующие директивы препроцессора:  
·       директивы компилятора #pragma, указывающие компилятору, как именно необходимо строить объектный код;  
·       директива включения файла **#include**, с помощью которой можно включить в текст программы текст из другого файла;

·       директивы условной компиляции

**#if**, **#else**, **#elif**, **#endif**, **#ifdef**, **#ifndef**, **defined**;  
·       директива определения лексем **#define**;  
·       директива отмены определения **#undef**

Директива **#include** может быть записана в одном из двух форматов:  
**#include** <*имяфайла*> и **#include** ”*имяфайла*”  
Первый формат позволяет включить имя файла из тех папок, которые заданы как стандартные для хранения включаемых файлов.Второй формат дает возможность записать произвольное имя файла.

Директива #**define** позволяет определить новые лексемы. Ее формат:  
#**define** *имялексемы [ (параметры) ] текстлексемы*  
**Лексемы с параметрами называют макроопределениями( макросами)**. Такие конструкции позволяют выполнить замещение лексем по-разному, в зависимости от фактических параметров.

**19.1 Рекурсии**

При решении многих задач пользуются методом сведения данной задачи к задаче, с меньшим числом предметов. Такой метод называют **методом рекуррентных соотношений.**

Существует класс задач, которые обладают таким свойством: вычисляя последующие значения интересующей нас величины, мы пользуемся только фиксированным числом последних значений.

Последовательности с таким свойством называют **рекуррентными**

**Рекурсивным** называется объект, который частично определяется через самого себя.

Максимальное число рекурсивных вызовов процедуры без возвратов, которое происходит во время выполнения программы, называется **глубиной рекурсии*.***

Число рекурсивных вызовов в каждый конкретный момент времени, называется ***текущим уровнем рекурсии.***  
В общем случае любая рекурсивная процедура включает в себя некоторое множество операторов S и один или несколько операторов рекурсивного вызова.

**Безусловные** рекурсивные процедура приводят к бесконечным процессам, так как практическое использо-вание процедур с бесконечным **самовызовом невозможно**.   
Такая невозможность вытекает из того, что для каждой копии рекурсивной процедуры необходимо выделял дополнительную область памяти, а бесконечной памяти не существует.

Следовательно, главное требование к рекурсивным процедурам заключается в том, что вызов рекурсивной процедуры должен **выполняться по условию,** которое на каком-то уровне рекурсии станет **ложным.**Если условие истинно, то рекурсивный спуск продолжается. Когда оно станет ложным, спуск заканчивается и начинается поочередный рекурсивный возврат из всех вызванных на данный момент копий рекурсивной процедуры.

**Формы рекурсивных процедур.**Структура рекурсивной процедуры может принимать три разные формы:  
 1. Форма с выполнением действий до рекурсивного вызова

**(с выполнениеем действий на рекурсивном спуске).**

void Rec();  
{S; if (*условие)* Rec();};

2.Форма с выполнением действий после рекурсивного вызова

**(с выполнением действий на рекурсивном возврате).** void Rec();  
{if (условие) Rec(); S;};

3. Форма с выполнением действий как до, так и после рекурсивного вызова **(с выполнением действий как на рекурсивном спуске, так и на рекурсивном возврате).**  
void  *Rec();*  
{ S1;  
if (*условие)* Rec();  
 S2;}

**Динамическое программирование** (ДП) — это вычислительный метод для эффективного решения задач с пересекающимися подзадачами. Возникло и сформировалось в 1950-53г. благодаря работам Р.Беллмана.

**Идея** **динамического программирования** состоит в разбиении задачи на несколько независимых подзадач, решении каждой из них, а затем вычислении результата.   
Для решения подзадач этот же алгоритм применяется рекурсивно или итерационно. При этом для каждой подзадачи запоми-нается вычисленный ответ, и если на каком-то шаге встретилась подзадача второй раз, то вычисления для нее не производятся.   
За счет большого количества пересекающихся подзадач это значительно уменьшает время работы.

В математике существуют числа **Фибоначчи.** Это числа следующего вида:1; 1 ; 2; 3; 5;……………  
Таким образом каждое следующее число получается как сумма двух предыдущих.Рекурсивное определение для чисел Фибоначчи :

**Рекурсивная функция вычисления чисел Фибоначчи  
int FibR(int n)  
{  
 if ((n == 1) || (n == 2))   
 return 1;  
 else  
 return FibR(n - 1) + FibR(n - 2);  
}**

**Рекурсивный НОД**

**int f\_nod( int x,y)  
 *{ рекурсивная функция}*  
 {  
 if (y>x) return f\_nod(y,x);  
 else if (y<=0) return x;  
 else return f\_nod(y,x mod y);  
 };**

**Двоичный поиск элемента в упорядоченной последовательности }  
void Search\_bin(int \*a, int l, int r,int x)** **int k,p,i,j;  
 { k= (l+r) / 2;  
 if (a[k]==x) return k;  
 else  
 { if (a[k] <=x)  
 l=k+1;  
 else r=k-1;  
 return Search\_bin (a,l,r,x); }}**

**{поиск наибольшего элемента элемента в последовательности }  
int mmax (int \*a, int i,int n)  
{**  
 **if (i==n) mmax=a[i];  
else  
 if ( (a[i]> mmax(a,i+1) && (i==n))   
 mmax=a[i];  
 else mmax= mmax(a,i+1,n);  
 }**

Это задача поиска **наибольшей общей последовательности, ( Longest Common Subsequence, LCS)**которая является подпоследовательностью нескольких последовательностей (обычно двух).

**Рекурсивное решение:   
int lcs\_length(char \* A, char \* B)** {  
 if (\*A == '\0' || \*B == '\0') return 0;  
 else if (\*A == \*B)  
 return 1 + lcs\_length(A+1, B+1);  
 else   
 return   
max(lcs\_length(A+1,B), lcs\_length(A,B+1));  
 }

**Итерационное решение** **наибольшей подпоследовательности**  
**char\* getLongComSub(char \*a, char \*b)** {  
 char \*\*max\_len,\*res;  
 max\_len= (char \*\*) malloc(strlen (a) + 1);  
 res= (char \*) malloc(strlen (b) + 1);  
  
 for(int i = 0; i <= strlen (a); i++)  
 max\_len[i]= (char \*) malloc(strlen (b) + 1);

for( i = strlen (a)- 1; i >= 0; i--)  
{  
 for(int j = strlen (b) - 1; j >= 0; j--)  
 { if(a[i] == b[j])  
 {  
 max\_len[i][j] = 1 + max\_len[i+1][j+1];  
 }  
 else  
 {max\_len[i][j] = max(max\_len[i+1][j], max\_len[i][j+1]);  
 }

}

}

k++;

i++;

j++;

**//продолжение наибольшей подпоследовательности**

int j,k=0;  
 for( i = 0, j = 0; max\_len[i][j] != 0; )  
 {  
 if(a[i] == b[j])  
 {  
 res[k]=a[i];  
 res[k+1]='\0';  
 k++; i++; j++;  
 }  
 else  
 { if(max\_len[i][j] == max\_len[i+1][j])  
 i++;  
 else j++; } }  
 return res;

}

**//Палиндром**

#include <stdlib.h>  
#include <string.h>  
void insert(int c,char \*A,int n)  
{  
 int i=0;  
 int len=strlen(A);  
 int t=len;  
 while(t>=n)

{

A[1+t]=A[t];  
 --t;  
 }  
 A[n]=c;

}

void pal(char \*st, int l,int r)  
{  
 int t,i;char ch;  
   
 if (l>=r)  
 return;  
 else  
 if (st[l]==st[r])  
 pal(st,l+1,r-1);  
 else

**// продолжение от палиндрома**

if (st[l]==st[r-1])  
 {  
 ch=st[r];  
 insert(ch,st,l);  
 proc(st,l+2,r-1);  
 }  
 else  
 {  
 ch=st[l];  
 insert(ch,st,r+1);  
 proc(st,l+1,r);  
 } }

void main ()  
{  
char x[100]="12345";  
proc(x, 0,5);  
puts (x);  
}

**Алгоритмы движения с заданными условиями**

Фишка может двигаться по полю длины N  только  вперед.  Длина хода фишки  не  более K.  Найти число различных путей,  по которым фишка может пройти поле от начала до конца. Обозначим через S(i) количество различных путей,  по  которым фишка может пройти поле от начала до позиции с номером i.  Предположим теперь, что для любого j от 1 до i известны значения величин S(j).   Задача  состоит  в  определении правила вычисления значения S(i+1),  используя значения известных величин. Легко заметить, что в  позицию  с  номером  i+1  фишка  может  попасть  из  позиций i, i-1,...,i-k. Следовательно, S(i+1)=S(i)+S(i-1)+...+S(i-k). Таким образом,  вычисляя  последовательно  значения   величин S(1), S(2),  ..., S(N) по описанному выше правилу, получаем значение S(N), которое и указывает общее количество различных путей, по которым фишка может пройти поле от начала до позиции с номером N.

**Покупка товара наибольшей стоимости**

Покупатель имеет купюры достоинством A(1), ...,A(n), а продавец - B(1), .. ,B(m). Необходимо найти максимальную стоимость товара Р, которую покупатель не может купить, потому что нет возможности точно рассчитаться за этот товар с продавцом, хотя денег напокупку этого товара достаточно.

Опишем процедуру вычисления S.  
 S:=0;  
 i:=1;  
 пока (i<=N) и (C[i]<=S+1)  
 нц  
 S:=S+C[i];  
 i:=i+1  
 кц  
 Если значение S не меньше суммарного количества денег покупателя, то покупатель может купить товар любой доступной ему стоимости, точно рассчитавшись за покупку. Иначе P=A[1]+...+A[N]-S.

1. **Задача Иосифа Флавия**

Пусть по кругу стоят n человек, начинают считать с 1-го, и каждый k-ый выбывает. Кто останется водить?  
 Итак, получается такая рекуррентная зависимость:

к=2:  
J(n) = 1,  
J(n) = 2J(n/2) - 1, если n четное,  
J(n) = 2J((n - 1)/2) + 1, если n нечетное.

Пример решения задачи с циклично связным списком:

struct people{int number; people \*next;};

void Step(people \*&top, int k)

{

k--;

for (k; k>1; top=top->next, k--);

people \*temp=top->next;

top->next=top->next->next;

top=top->next;

delete temp;

}

**//Продолжение Иосифа Флавия**

void main()

{

setlocale(LC\_ALL,".1251"); int n, k;

cout<<"Введите количество людей в группе?\n";

cin>>n;

cout<<"Какого по счету удаляем?\n";

cin>>k;

if (k==1)

{

cout<<"В живых останется человек с номером "<<n<<endl; return;

}

people \*top=new people;

top->number=1;

people \*last=top;

for (int i=2; i<=n; i++)

{

people \*p=new people;

p->number=i;

last->next=p;

last=p;

}

last->next=top;

for (int i=1; i<n; i++)

Step(top, k);

cout<<"В живых останется человек с номером "<<top->number<<endl;

delete top;

}

1. **Задача о рюкзаке**

Имеются *m* предметов с номерами от 0 до *m*-1, для каждого из которых известна масса в килограммах *pj* и стоимость *cj* (*j* = 0,1,…,*m*-1). Определить, какие предметы необходимо положить в рюкзак, чтобы их общая масса не превышала P кг, а общая стоимость была максимальной.  
Сколько предметов каждого типа он должен положить в рюкзак, чтобы суммарная ценность снаряжения была максимальной при ограничении на массу рюкзака?

**Жадный алгоритм** для задачи о рюкзаке состоит в следующем:   
Выбрать максимально дорогой предмет, стоимости *Cmax* . Упорядочить предметы по «удельной стоимости» (стоимости деленной на вес), и набивать рюкзак наиболее «удельно дорогими» предметами, пока они влезают. Пусть стоимость этого решения *Cgreedy* В зависимости от того, что больше, *Cmax* или *Cgreedy*, выбрать первое или второе решение.Задача укладки рюкзака сложна. Если перебирать всевозможные подмножества данного набора из *k* предметов, то получится решение сложности не менее чем *O*(2k). Мы рассмотрим решение данной задачи для случая, когда все входные данные - целочисленные, сложность которого будет *O*(*kW*). Рассмотрим следующую функцию. Пусть R(*s*, *n*) есть максимальная стоимость предметов, которые можно уложить в рюкзак максимальной вместимости *n*, если можно использовать только первые *s* предметов из заданных *k*. Зададим краевые значения функции R(*s*, *n*). Если *s* = 0, то R(0, *n*) = 0 для всех *n* (ни один предмет нельзя брать, поэтому максимальная стоимость равна 0). Если *n* = 0, то R(*s*, 0) = 0 для всех *s* (можно брать любые из первых *s* предметов, но вместимость рюкзака равна 0).

**// О рюкзаке**

Cоставим рекуррентное соотношение: необходимо из предметов с номерами 1, ..., *s* составить рюкзак максимальной стоимости, чей вес не превышает *n*. При этом возможно два случая: когда в максимальный рюкзак включен предмет с номером *s* и когда предмет *s* не попал в максимальный рюкзак. Если предмет *s* не попал в максимальный рюкзак массы *n*, то максимальный рюкзак будет составлен только из предметов с номерами 1, ..., *s* - 1, следовательно, *A*(*s*, *n*) = *A*(*s* - 1, *n*). Если же в максимальный рюкзак включен предмет *s*, то масса оставшихся предметов не превышает *n* - *w*s, а от добавления предмета *s* общая стоимость рюкзака увеличивается на *p*s. Значит, *A*(*s*, *n*) = *A*(*s* - 1, *n* - *w*s) + *p*s. Теперь из двух возможных вариантов составить рюкзак массы <= *n*, из предметов 1, ..., *s* нужно выбрать :

***A*(*s*, *n*) = max (*A*(*s* - 1, *n)*,  
 *A*(*s* - 1, *n* - *w*s) + *p*s.**

**Ханойские башни**

Есть три стержня A, B, и C. На стержень A надето N дисков, наверху самый маленький, каждый следующий диск больше предыдущего, а внизу самый большой. На другие стержни дисков не надето.   
Hеобходимо перенести диски со стержня A на стержень C, пользуясь стержнем B, как вспомогательным, так, чтобы диски на стержне C располагались в том же порядке, в каком они располагаются на диске A перед перемещением.   
При перемещении никогда нельзя класть больший диск на меньший.

void Move(int n, char x,y,z);  
//х - А y - В z - C  
{ if (n==1) scanf(“ диск 1 %d -> %d“ ,x,y);  
 else  
 if (n>1)  
 {  
 Move(n-1,x,z,y);  
 printf(“диск %2d %d ->%d”,n,x,y);  
 Move(n-1,z,y,x);  
}

**Метод быстрой сортировки Хоара**

Метод быстрой сортировки массива разрботан в 1962 году профессором Оксфордского университета К. Хоаром. **Алгоритм** состоит в том, чтобы, выбрав некоторый элемент разделяющим, переместить все элементы меньшие (или равные) его левее, а большие – правее. После этого применить тот же ход для левой части и правой части массива, если в этой части больше одного элемента. Алгоритм – рекурсивный, т.е. соответствующая процедура обращается при реализации к самой себе. Сложность алгоритма – O(n\*logn) – в среднем. В худшем случае (когда после разделения на каждом шаге один из подмассивов состоит из одного элемента, а второй – из оставшихся) сложность O(n2)

void qsort(int a[], int l, int r)  
{  
int i = l, j = r, p;  
int x = a[(l + r)/2];  
while (i <= j)  
{

while (a[i] < x)   
 ++i;  
while (x < a[j])

--j;

if (i <= j)  
 { p = a[i]; a[i] = a[j]; a[j] = p; i++; j--; }

}  
if (l < j)  
 qsort(a, l, j);  
if (i < r)  
 qsort(a, i, r);

}

**Рекурсивные функции сортировки**

Упорядочить элементы последовательности A=(ai), i=1..n, используя рекурсивную функцию сортировки.  
void rec\_sort(int \*a, int i, int n) // рекурсивной функция сортировки вставками  
{ if (i < n)  
{ int max\_el = a[i], max\_ind = i;  
for (int j = i; j < n; ++j)  
if (a[j] > max\_el)  
{ max\_el = a[j]; max\_ind = j;}

if (max\_ind > i)  
{ a[i] ^= a[max\_ind]; a[max\_ind] ^= a[i]; a[i] ^= a[max\_ind];}  
rec\_sort(a, i+1, n);  
}}

**Решение комбинаторных задач: генерация k-элементных подмножеств, генерация перестановок**

Генерация k-элементных подмножеств  
В комбинаторике такие подмножества называют сочетаниями из *n* элементов по *k* элементов и обозначают *Cnk* , при программировании гораздо удобнее использовать следующие рекуррентные соотношения

****

Обычно генерацию всех k-элементных подмножеств проводят в лексикографическом порядке, тем более что в данном случае это не приводит ни к усложнению алгоритма, ни к увеличению его вычислительной трудоемкости. Порядок подмножеств называется лексикографическим, если для любых двух подмножеств справедливо, что ранее должно быть сгенерировано то из них, из индексов элементов которого можно составить меньшее k-значное число в n-ричной системе счисления (или в десятичной, для n < 10). Идея сведения данной задачи к задаче меньшей размерности следующая. Первым элементом подмножества может быть любой элемент, начиная с первого и заканчивая (n – k + 1)-м элементом. После того, как индекс первого элемента подмножества зафиксирован, осталось выбрать k – 1 элемент из элементов с индексами, большими, чем у первого. Далее поступаем аналогично. Когда выбран последний элемент, то мы достигли конечного уровня рекурсии и выбранное подмножество можно обработать.

1. **Классическая архитектура компьютера**

Архитектура компьютера - логическая организация, структура и ресурсы компьютера, которые может использовать программист.   
Архитектура определяет принципы действия, информационные связи и взаимное соединение основных логических узлов компьютера.

Принципы построения компьютеров (принципы фон Неймана):

1.Принцип программного управления.  
2. Принцип однородности памяти.   
3. Принцип адресности.

Основные компоненты архитектуры персонального компьютера:

**Процессор** **Внутренняя (основная) память** **Внешняя память**

**Устройство ввода** **Устройство вывода**

Внутренняя память обладает 2мя основными свойствами: дискретность, адресуемость. Поле памяти, с которым может работать команда, характеризуется адресом(числовое значение, определяющее местоположение поля в памяти) и содержимым(числовое значение, хранящееся в этом поле).

1. **Процессор - программно-управляемое устройство**

В общем случае центральный процессор - это основной рабочий компонент компьютера, который выполняет арифметические и логические операции, заданные программой, управляет вычислительным процессом и координирует работу всех устройств компьютера. содержит:   
- арифметико-логическое устройство;   
- шины данных и шины адресов;   
- регистры; - счетчики команд;   
- кэш - очень быструю память малого объема;   
- математический сопроцессор чисел с плавающей точкой

**Команды процессора.**

Каждая команда представляют собой последовательность двоичных разрядов и содержит:  
 код операции, который надо выполнить по данной команде,

информацию об операндах,  
 информацию о результате.  
Операнд – элемент информации, участвующей в выполняемой операции.

Перед выполнением программа загружается в некоторую область основной памяти. Перед запуском программы процессору сообщается адрес первой команды, с которой надо начинать выполнение программы, затем последовательно выполняются следующие действия:

1)      Извлечение из основной памяти команды, адрес которой находится в счетчике команд.  
2)      Расшифровка.  
3)      При необходимости операнды извлекаются из памяти.  
4)      Выполнение операции с кодом, заложенным в данной команде (если результат операции не надо помещать в память, то процессор переходит к действию 6)  
5)      Сохранение результата в памяти.  
6)      Вычисление адреса следующей команды и запоминание его в счётчике команд. Переход к действию 1

**Системы счисления и представления в памяти компьютера числовых и символьных данных**

Система счисления (с/с)– совокупность приемов записи и наименования чисел. Если в системе счисления значение каждой цифры в записи числа зависит от того места, на котором она стоит в записи относительно расположения запятой, то эта система счисления называется ПОЗИЦИОННОЙ.

Десятичная с/с, в которой каждый символ числа – это одна из десятичных цифр: **0, 1, 2, 3, 4, 5, 6, 7, 8, 9.** Число 10 - основанием десятичной системы счисления.

В двоичной системе счисления каждая цифра – это двоичное число, то есть, 0 или 1. Число 2 здесь является основанием двоичной системы счисления.

В троичной системе счисления каждая цифра – это троичное число, то есть, 0, 1 или 2. Число 3 здесь является основанием троичной системы счисления.

В восьмеричной системе счисления каждая цифра – это восьмеричное число, то есть, 0, 1, 3, 4, 5, 6 или 7.Число 8 здесь является основанием троичной системы счисления.

В шестнадцатиричной системе счисления каждая цифра – это шестнадцатиричное число (здесь к десятичным цифрам 0, 1, 3, 4, 5, 6, 7, 8, 9 добавляются первые буквы латинского алфавита - для обозначения чисел 10=A, 11=B, 12=C, 13=D, 14=E, 15=F).Число 16 здесь является основанием шестнадцатиричной системы счисления

**Представление в памяти компьютера числовых и символьных данных**

Любая информация (числовая, текстовая, звуковая, графическая и т.д.) в компьютере представляется (кодируется) в так называемой двоичной форме. Как оперативная, так и внешняя память, где и хранится вся информация, могут рассматриваться, как достаточно длинные последовательности из нулей и единиц, единицей измерения информации является бит, более крупными единицами измерения информации являются байт, килобайт и т.д. Один байт состоит из восьми бит, а каждая последующая величина больше предыдущей в 1024 раза. Байта достаточно для хранения 256 различных значений, что позволяет размещать в нем любой из алфавитно-цифровых символов, если только мы можем ограничиться языками с небольшими алфавитами типа русского или английского. Первые 128 символов (занимающие семь младших бит) стандартизированы с помощью кодировки ASCII (**American Standart Code for Information Interchange).**

**Представление чисел в Оперативной памяти.**

При представлении чисел в памяти, рассматривают множество целых Z и действительных R чисел

**Запись чисел с фиксированной точкой.**

Целые числа при представлении в памяти компьютера записываются как числа с фиксированной точкой, точка (запятая) обычно зафиксирована в конце разрядной сетки. Обычно целые числа занимают в памяти компьютеров один, два, четыре байта или восемь байт. Один, как правило, старший бит отводится под знак числа. Знак положительного числа "+" кодируется нулем, а знак отрицательного числа "-" - единицей. Целые положительные числа представляются в так называемом **двоичном прямом коде**. Целые отрицательные числа представляются в так называемом **двоичном дополнительном коде**.

**Запись отрицательных чисел с фиксированной точкой.**

Отрицательные числа всегда имеют в своем двоичном представлении **единицу в самом старшем разряде**, который поэтому называют знаковым, а абсолютная величина кодируемого числа получается как двоичное дополнение остальных бит (нули нужно заменить на единицы и наоборот), увеличенное на один. Отрицательные числа представляются в дополнительном двоичномкоде, так называемом дополнении, которое получается в результате:

1) так называемого инвертирования двоичного представления модуля исходного числа, когда нулевые биты превращаются в единичные, а единичные - в нулевые;

2)добавления к инвертированному представлению единички.

**Запись положительных чисел с фиксированной точкой.**

В каждом бите при представлении данных можно записать 0 или 1 .

При таком кодировании каждый двоичный разряд представляет собой степень цифры 2:

При этом минимально возможное число, которое можно записать таким двоичным кодом, равно 0.

Максимально возможное число, которое можно записать таким двоичным кодом, можно определить как: М= 2n -1.

**Двухбайтовое представление** числа с фиксированной точкой получается следующим образом:

Берется СЛОВО или ДВА байта (16 битов) и рассматривается как единое целое. Реально в ОП СТАРШИЙ байт МЕНЯЕТСЯ местами с МЛАДШИМ внутри этого СЛОВА.

**Четырехбайтовое число** с фиксированной точкой характеризуется тем, что наряду с тем, что в каждом слове перевернуты байты, сами слова внутри этого двойного слова перевернуты.

**Восьмибайтовое число** с фиксированной точкой характеризуется тем, что наряду с тем, что в каждом слове перевернуты байты, сами слова внутри этого двойного слова перевернуты.



**Представление чисел с плавающей точкой**

Числа с плавающей точкой представляются в виде так называемых мантиссы и порядка. Число с *плавающей точкой* это есть способ записи числа в позиционной системе счисления с основанием R, когда некоторое число N представляется в виде N = q \* Rp, p - целое число, называемое порядком числа, q - правильная дробь, называемая мантиссой. Если q < 0.1, то мантисса называется НЕНОРМАЛИЗОВАННОЙ, в противном случае - НОРМАЛИЗОВАННОЙ. Представлением десятичного числа в виде числа с плавающей точкой в оперативной памяти ПЭВМ называется представление, получаемое следующим образом:

1. Исходное десятичное число с точкой переводится в число в шестнадцатеричной системе счисления.

2. Полученное шестнадцатеричное представление числа с точкой переводится в двоичную систему счисления.

3. Полученное двоичное представление числа с точкой записывается в двоичной системе счисления посредством применения так называемых усеченной мантиссы и усеченного порядка

4. Определяется характеристика числа по формуле 7F + значение усеченного порядка

5. В первом слева бите четырехбайтового представления числа кодируется знак числа (0 – если число положительное, 1 – если число отрицательное ).

6. В следующих восьми битах, если смотреть на них слева направо, записывается характеристика.

7. Оставшиеся биты ( 32 - 1 - 8 = 23 ) заполняются усеченной мантиссой.

8. Полученное на предыдущих шагах четырехбайтовое представление при расположении его в памяти хранится там в перевернутом виде

1. **Регистры, назначение и характеристика**

Регистр является устройством временного хранения данных и используется с целью облегчения арифметических, логических и пересылочных операций. Регистры 32-битных моделей IA-32 являются расширением регистров 16-битных моделей семейства.

Регистры общего назначения

Восемь регистров общего назначения имеют длину в 32 бит и содержат адреса или данные. Они поддерживают операнды-данные длиной 8, 16, 32 и (при использовании двух регистров) 64 бит. Эти регистры называются **EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP**.В машинных кодах никаких букв нет, и регистры кодируются тремя битами (в указанном выше порядке - от 000 до 111). Доступ к младшим 16 бит этих регистров выполняется независимо при использовании соответствующих имен 16-битных регистров: AX, BX, CX, DX, SI, DI, BP и SP. Также могут использоваться индивидуально младший (биты 0-7) и старший (биты 8-15) байты регистров AX, BX, CX, DX. Им соответствуют обозначения AH, DH, CH, BH и AL, DL, CL, BL.

**EAX** - аккумулятор, операнд-источник или приемник результата (некоторые инструкции могут быть короче на один байт при использовании EAX);   
**EBX** - указатель на данные в сегменте DS;   
**ECX** - используется в качестве счётчика (в циклах) и для цепочечных и циклических (с префиксом REP и т.п.) инструкций;

**EDX** - регистр данных (подобно eax) (адрес порта ввода-вывода для инструкций IN/INS, OUT/OUTS; )  
**ESI** - указатель на операнд-источник;   
**EDI** - указатель на операнд-приемник;   
**ESP** - указатель на вершину стека.  
**EBP** - указатель на данные в сегменте SS.

**При записи ячеек памяти (переменных) в командах пишут обозначения**:   
byte ptr – 8-разрядная переменная, – эти слова, записанные ПЕРЕД адресом области памяти, говорят, что в операции над этой памятью должно участвовать данное, расположенное в одном байте   
word ptr – 16-разрядная переменная, - эти слова, записанные ПЕРЕД адресом области памяти, говорят, что в операции над этой памятью должно участвовать данное, расположенное в двух байтах (в слове)  
dword ptr – 32-разрядная переменная, - эти слова, записанные ПЕРЕД адресом области памяти, говорят, что в операции над этой памятью должно участвовать данное, расположенное в четырех байтах (в двойном слове)  
qword ptr – 64-разрядная переменная  
tbyte ptr – 80-разрядная переменная

**Флаги процессора**

Есть еще два регистра: это регистр флагов EFLAGS и указатель инструкций EIP. Состояние регистра флагов может меняться после каждой инструкции в зависимости от полученного результата. Регистр EIP содержит адрес начала следующей инструкции в памяти. Его значение увеличивается каждый раз, когда из памяти извлекается для исполнения очередная инструкция, на величину размера этой инструкции.

**Указатель команд**   
Указатель команд (EIP) является 32-разрядным регистром. Он содержит смещение следующей команды, подлежащей выполнению. Относительный адрес отсчитывается от начала сегмента исполняемой задачи. Указатель команд непосредственно недоступен программисту, но он управляется явно командами управления потоком, прерываниями и исключениями , командами условного перехода. Младшие 16 бит регистра EIP обозначаются IP и могут быть использованы процессором независимо при исполнении 16-битного кода.

**ZF** – флаг нуля. Устанавливается в 1, если результат предыдущей операции – ноль   
**SF** – флаг знака. Он всегда равен старшему биту результата   
**CF** – флаг переноса. Устанавливается в 1, если результат предыдущей операции над беззнаковыми числами не уместился в приёмнике и произошёл перенос из старшего бита, или если требуется заём (при вычитании), иначе 0   
**OF** – флаг переполнения. Устанавливается в 1, если результат предыдущей арифметической операции над числами со знаком выходит за допустимые для них пределы

**AF** – флаг полупереноса. Устанавливается в 1, если в результате предыдущей операции произошёл перенос или заём из третьего бита в четвёртый. Этот флаг используется автоматически командами двоично-десятичной коррекции.   
**PF** – флаг чётности. Устанавливается в 1, если младший байт результата предыдущей команды содержит чётное число битов, равных единице, иначе 0.   
**IF** – флаг прерываний. 1 – прерывания разрешены, 0 – прерывания запрещены   
**DF** – флаг направления. 0 – строки обрабатываются в сторону увеличения адресов, 1 – в сторону уменьшения адресов

Сегментные регистры   
Микропроцессор 80386 включает шесть непосредственно доступных 16-битных регистров селекторов сегментов. С каждым сегментным регистром ассоциирован программно-недоступный кэш дескриптора соответствующего сегмента, содержащий базовый адрес сегмента в линейном адресном пространстве, предел сегмента и атрибуты сегмента: CS, SS , DS, ES, FS, GS. Регистр CS хранит селектор сегмента кода. Регистр SS хранит селектор сегмента стека.   
Регистры DS, ES, FS и GS хранят селекторы сегментов данных. Если инструкция обращается к памяти, но содержит только смещение, то считается, что она обращается к данным в сегменте DS. Сегмент ES может использоваться без явного указания в цепочечных командах. Сегменты FS и GS используются при обращении к памяти только при явном использовании в инструкции префиксов этих сегментов.

Также существуют регистры управления сегментированной памятью, регистры управления, регистры отладки.

**Способы адресации.**

КОП РР,РР Регистровая адресация   
КОП РР,ОП Прямая адресация  
 КОП ОП,РР Косвенная адресация  
КОП РР,НД Непосредственная адрес-я   
КОП ОП,НД

Регистровая адресацияОперанды могут располагаться в любых регистрах общего назначения и сегментных регистрах. Для этого в тексте программы указывается название соответствующих регистров, длины регистров должны совпадать

Непосредственная адресация

Некоторые команды позволяют указывать один из операндов непосредственно в тексте программы. Например: **mov d,100.** При такой адресации один операнд находится в регистре или в памяти, а второй записан в самой команде, причем ДЛИНА этих данных тоже ВСЕГДА ОДИНАКОВА! В качестве НЕПОСРЕДСТВЕННОГО ДАННОГО может выступать десятичное число, шестнадцатеричное число, символ и двоичное число.

Прямая адресация

Если у операнда, располагающегося в памяти, известен адрес, то его можно использовать. **mov eax,x; mov ax,i** ; **mov al,c**

Косвенная адресация

По аналогии с регистровыми и непосредственными операндами адрес операнда в памяти также можно не указывать, а хранить в любом регистре. Адрес операнда можно считывать из **ЕАХ, ЕВХ, ЕСХ, EDX, ESI, EDI, ЕВР и ESP,ВХ, SI, DI ,ВР.**

Косвенная адресация включает в себя следующие :

**Индексную базовую базово-индексную**

**индексная**

При задании адреса используется один из индексных регистров  **esi ( si ) или edi ( di ).** При таком способе адресации адрес области памяти может иметь вид:

-индексный регистр в квадратных скобках. Например, команда вида **mov [esi], ’\0’**

-некоторое символическое имя и индексный регистр в квадратных скобках. **mov arr[esi], 10**   
-в квадратных скобках индексный регистр, символическое имя и константы, соединенные знаком **+. mov [arr + esi], 1 mov [arr + esi+4], 2**

С помощью этого метода разрешается организовывать доступ к одномерным массивам байтов. Смещение соответствует адресу начала массива. Число в регистре - индексу элемента массива, который надо считать. Eсли массив состоит из байтов, базовый регистр увеличиваем на 1, если из слов, увеличиваем на 2, если из двойных слов - на 4. Масштабирование, которое выполняется в Си при увеличении адреса можно заменить умножением базового регистр на 2, для двойных слов - на 4.

**базовая**

Если применяется базовая адресация, то при задании адреса используется один из базовых регистров: либо **ЕBX(ВХ)** либо **BP**

При таком способе адресации адрес области памяти, куда помещается информация, может быть записан:

-заключением в квадратные скобки базового регистра: MOV [ebx],10  
-заключением в квадратные скобки не базового регистра, символического имени: MOV [x+ebx],10

- заключением в квадратные скобки базового регистра, символического имени и некоторого числа-смещения: MOV [x+ebx+4],10

С помощью этого метода разрешается организовывать доступ к одномерным массивам байтов. Смещение соответствует адресу начала массива. Число в регистре - индексу элемента массива, который надо считать.

**базово-индексная**

это когда при задании адреса используются один из базовых регистров : **ebx (bx) ebp (bp)** и один из индексных регистров:  **esi (si)** либо **edi (di)**

-либо заключением в квадратные скобки базового и индексного регистров, например, можно записать так : **MOV[BX,SI],25**  
- либо заключением в квадратные скобки не только базового и индексного регистров, но и некоторого символического имени, например **MOV А[ЕBX,ЕSI],2007**   
- либо заключением в квадратные скобки не только базового регистра и индексного регистра, некоторого символического имени, но и некоторого числа, называемого смещением. Например,

**MOV [X+ЕBX+ЕSI+4],2007**

mov ax,[bx+si+2]   
mov ax,[bx][si]+2   
mov ax,[bx+2][si] **Разные способы записи одного и того же**

mov ax,[bx][si+2]   
mov ax,2[bx][si]

адресация по базе с индексированием и масштабированием

Это самая полная схема адресации, в которую входят все случаи, рассмотренные ранее как частные. Смещение может быть байтом или двойным словом. Если ESP или ЕВР используются в роли базового регистра, селектор сегмента операнда берется по умолчанию из регистра SS, во всех остальных случаях - из DS.

**mov [ebx-4][esi\*4],edx**

Арифметические команды

|  |  |  |
| --- | --- | --- |
| Двоичной aрифметики | | Десятичной арифметики |
| Add | Imul | Aaa |
| Adc | Mul | Daa |
| Inc | Idiv | Aas |
| Sub | Div | Das |
| Sbb | Neg | Aam |
| Dec |  | Aad |

**Команды сложения**

**add приёмник, источник** – выполняет сложение приёмника и источника, результат заносится в приёмник. Источник не изменяется, меняются флаги. Команда ADD не различает числа со знаком и без знака, но, употребляя значения флагов CF (перенос при сложении чисел без знака), OF (перенос при сложении чисел со знаком) и SF (знак результата), разрешается применять ее и для тех, и для других.

**adc приёмник, источник** – выполняет сложение приёмника, источника и флага CF. Обычно эта команда используется для сложения чисел повышенной точности.

**inc приемник (инкремент) - у**величивает приемник (регистр или переменная) на 1. Единственное отличие этой команды от **add приемник,1**состоит в том, что флаг CF не затрагивается, а остальные меняются в соответствии с правилами сложения.

**Команды вычитания**

**sub приёмник, источник** – вычитает источник из приёмника, результат заносит в приёмник.  
**sbb приёмник, источник** – вычитает из приёмника значение источника, затем вычитает значение CF . Её можно использовать для вычитания 64-битных слов.

**dec приемник (декремент) - у**величивает приемник (регистр или переменная) на 1. Единственное отличие этой команды от **sub приемник,1**состоит в том, что флаг CF не затрагивается, а остальные меняются в соответствии с правилами сложения.

**Команды умножения**

**MUL источник - умножение чисел без знака**  
Источник умножается на **al|ax|eax** в зависимости от разрядности операнда.  
 Результат располагается в **ax|dx:ax|edx:eax** соответственно.Если результат поместился в младшую половину, то CF=OF=0, иначе CF=OF=1. ZF, SF, AF, PF – не определены.

Умножение со знаком:

1.**IMUL источник** (регистр или переменная) умножается на AL, АХ или ЕАХ (в зависимости от размера операнда), и результат располагается в АХ, DX и AX или EDX и EAX соответственно.  
2.**IMUL приемник, источник:** источник (число, регистр или переменная) умножается на приемник (регистр), и результат заносится в приемник.  
3.**IMUL приемник, источник1, источник2:**  источник 1 (регистр или переменная) умножается на источник 2 (число), и результат заносится в приемник (регистр).

1. Команды деления

**DIV источник – целочисленное деление без знака.**  
 **IDIV источник – целочисленное деление co знаком.**  
Команда DIV выполняет деление целого числа без знака , из регистра АХ (в случае деления на байт), DX и AX (в случае деления на слово) и EDX и EAX (в случае деления на двойное слово), на операнд-источник (целое число без знака). Частное заносится в AL(в случае деления на байт), AX(в случае деления на слово),EAX(в случае деления на двойное слово); остаток в AH(в случае деления на байт), DX(в случае деления на слово), EDX(в случае деления на двойное слово). Размер делимого в два раза больше размеров делителя и остатка. Флаги ZF, SF, CF, OF, AF, PF после этой команды не определены.

**Остальные команды**

Команда изменения знака NEG. **NEG приемник (регистр или переменная)**Выполняет над числом, содержащимся в приемнике (регистр или переменная), операцию дополнения до двух. Эта операция эквивалентна обращению знака операнда, если рассматривать его как число со знаком. Если приемник равен нулю, флаг CF = 0, иначе 1. Остальные флаги (OF, SF, ZF, AF, PF) назначаются в соответствии с результатом операции.

Команда XCHG выполняет обмен двух значений. Обмен значений можно выполнять двух между регистрами или между регистрами и памятью:

**XCHG приемник, источник**

Команда NOP - ничего не делающая команда

**NOP**

Эта команда не делает абсолютно ничего (пустая команда). Она только занимает пространство и время. Используется для резервирования места в сегменте кода или организации программной задержки.

22. **Команды преобразования:**

**CBW Преобразование байта в слово (convert byte to word)**Команда CBW заполняет регистр AH знаковым битом содержимого регистра АL, преобразуя тем самым 8-разрядное число со знаком в 16-разрядное. Команду удобно использовать для преобразования байтового делимого в двухбайтовое (слово) при делении на 8-разрядный операнд.

**CWD Преобразование слова в двойное слово**

**(convert word to double** **word)**

Команда CWD заполняет регистр DX знаковым битом содержимого регистра АХ, преобразуя тем самым 16-разрядное число со знаком в 32-разрядное. Команду удобно использовать для преобразования двухбайтового делимого в четырехбайтовое (двойное слово) при делении на 16-разрядный операнд.

**CWDE Преобразование слова в двойное слово**Команда CWDE заполняет старшие два байта регистра EAX знаковым битом содержимого регистра АХ, преобразуя тем самым 16-разрядное число со знаком в 32-разрядное.

**CDQ Преобразование двойного слова в четвертное слово**

**(convert double to quadric)**Команда CDQ заполняет регистр EDX знаковым битом содержимого регистра EАХ, преобразуя тем самым 32-разрядное число со знаком в 64-разрядное. Команду удобно использовать для преобразования четырехбайтового делимого в восьмибайтовое (четвертное слово) при делении на 32-разрядный операнд.

**23Команды сравнения:**

**Логическое сравнение**

**TEST приемник, источник**Команда используется для логического умножения двух операндов (вычисляет результат действия побитового «логического И») над приемником и источником, не сохраняя результата.  
**TEST ECX,EDX**

Устанавливает флаги SE ZF и PF в соответствии с полученным показателем, флаги OF и CF обнуляются, значение AF не определено.  
 TEST, так же как и CMP используется в основном в сочетании с командами условного перехода и условной пересылки данных.

Команда TEST устанавливает только флаги, а операнд не изменяется.

**Команды сравнения**

**CMP приемник, источник**  
Сравнивает приемник и источник и устанавливает флаги. Сравнение - путем вычитания источника из приемника, причем результат вычитания никуда не записывается.  
 Единственным следствием работы этой команды оказывается изменение флагов CF, OF, SF, ZF, AF и PF. Обычно команду СМР используют вместе с командами условного перехода и условной пересылки данных .

**если ( АХ ) < ( BX ) , то значение бита SF=1.**  
 **если ( АХ ) > ( BX ) , то значение бита SF=0**  
 **если ( АХ ) = ( BX ) , то значение бита ZF=1.**

24 Команды управления:

**Команда безусловного перехода  
JMP операнд**  
Команда передает управление в другую точку программы, не сохраняя информации для возврата. Операндом может быть непосредственный адрес для перехода (обычно метка), а также регистр или переменная, содержащая адрес. Команда не изменяет содержимое флагов.  
Общий вид команды

**JMP ADDRESS**

**Команды условных переходов**Команды условного перехода имеют следующий общий вид :  
 **КОП Адрес** КОП определяет условие перехода.  
Адрес определяет, куда нужно перейти, если условие, закодированное в коде операции, соответствует текущему значению регистра флажков.

Различают команды условного перехода для знаковых и беззнаковых чисел.   
Слова «выше -**A**» и «ниже - **B**» в таблице относятся к сравнению чисел без знака; слова «больше - **G**» и «меньше - **L**» учитывают знак.  
!!Команды условных переходов не поддерживают дальних переходов (>128)

**Для чисел без знака  
JB/JNAE**

Переход если ниже; переход если не выше или равно **CF = 1**

**JAE/JNB**

Переход если выше или равно; переход если не ниже  **CF = 0**

**JBE/JNA**

Переход если ниже или равно; переход если не выше **CF=1** **или ZF = 1**

**JA/JNBE**

Переход если выше; переход если не ниже или равно **CF=0 и ZF=0**

**Для чисел со знаком  
JL/JNGE**

Переход если меньше; переход если не больше или равно **S<>0**

**JGE/JNL**

Переход если больше или равно; переход если не меньше **C = 0**

**JLE/JNG**

Переход если меньше или равно; переход если не больше **Z=1 или**

**Z<>0**

**JG/JNLE**

Переход если больше; переход если не меньше илиравно **Z=0 или S=0**

**Для прочих данных   
JE/JZ** Переход если равно **ZF = 1**

**JNE/JNZ** Переход если не равно **ZF = 0**

**JP** Переход при успешной проверке четности **JPE**;

переход при четности **PF = 1**

**JNP**  Переход при неуспешной проверке четности  
**JPO** Переход при нечетности **PF = 0**   
**JS**  Переход если есть знак **SF = 1**

**JNS**  Переход если нет знака **SF = 0**   
**JC**  Переход если есть перенос **СF = 1**   
**JNC** Переход если нет переноса **CF = 0**  
**JO**  Переход если есть переполнение **OF = 1**  
**JNO** Переход если нет переполнения

**Команда перехода с учетом состояния регистра CX (ЕСХ)  
JCXZ метка**

Выполняет ближний переход на указанную метку, если **СХ=0   
JECXZ метка**

Выполняет ближний переход на указанную метку, если **ЕСХ=0**

**25 Команды организации циклов:**

для предотвращения выполнения цикла при нулевом ecx или cx нужна команда jecxz/jcxz. Если этого не сделать, то при изначально нулевом ecx/cx цикл повторится 4 294 967 295 или 65 536 раз;

**Команда организации цикла со счетчиком.**  
 **LOOP метка**

1. От содержимого регистра CX (ЕСХ) отнимается 1 и результат снова помещается в регистр CX(ЕСХ)  
2. Если содержимое регистра CX(ЕСХ) оказывается равным 0, то больше ничего не делается, а точнее, выполняется следующая за командой LOOP команда.  
3.Если же содержимое регистра CX(ЕСХ) не равно 0, то передается управление на команду, которая помечена меткой, указанной в команде LOOP

Эта команда используется для организации циклов, в которых регистр ECX (CX) играет роль счетчика.

**Команда организации цикла с учетом флага ZF.**  
Имеются разновидности команды LOOP:  
 **LOOPE** метка цикл, пока равно  
 **LOOPZ** метка цикл, пока ноль  
 **LOOPNE** метка цикл, пока не равно  
 **LOOPNZ** метка цикл, пока не ноль

Проанализировать регистр ZF:  
1.От содержимого регистра CX (ЕСХ) отнимается 1 и результат снова помещается в регистр CX  
2.Если содержимое регистра CX(ЕСХ) =0, выполняется следующая за командой LOOP команда.  
3.Если содержимое регистра CX(ЕСХ) не равно 0, то передается управление на команду, которая помечена меткой, указанной в команде LOOP .

4. Если регистр ZF=0, то для команд LOOPE и LOOPZ это означает выход из цикла, а для команд LOOPNE и LOOPNZ – переход к началу цикла.  
5.  Если регистр ZF=1, то для команд LOOPE и LOOPZ это означает переход к началу цикла, а для команд LOOPNE и LOOPNZ – выход из цикла.  
При этом надо учитывать, что сами команды флаг ZF не меняют, поэтому флаг должен быть установлен предыдущей командой

**26 Команды пересылки:**

**Команда пересылки данных.  
MOV приемник, источник**Команда MOV действует аналогично операторам присваивания из языков высокого уровня, то есть **mov ах, bх ~ ах = bх**

**Условная пересылка данных.**Условная пересылка данных

(команды появились в процессорах PentiumPro и Pentium II).  
 **CMOVcc приемник,источник**  
Это набор команд, которые копируют содержимое источника в приемник, в зависимости от значения тех или иных флагов из регистра FLAGS  
Источник - регистр общего назначения или переменная, а приемником - только регистр.

**Команда пересылки с расширением знака  
 MOVSX приемник, источник**  
 Копирует содержимое источника (регистр или переменная размером в байт или слово) в приемник (16- или 32-битный регистр) и расширяет знак аналогично командам CBW/CWDE

**Команда пересылки с расширением нулями  
 MOVZX приемник, источник**  
 Пересылка с расширением нулями. Копирует содержимое источника (регистр или переменная размером в байт или слово) в приемник (16- или 32-битный регистр) и расширяет нулями.





**27 Логические команды:**

логическое И - and, логическое ИЛИ - or,

Исключающее ИЛИ - xor;   
NOT инвертирует бит источника.  
логические, арифметические и циклические сдвиги;   
проверка битов и операндов;   
установка и очистка битов (флагов) регистра состояния процессора

( PSW).

**циклические**

Циклические сдвиги позволяют сдвигать биты кода операнда по кругу (по часовой стрелке при сдвиге вправо или против часовой стрелки при сдвиге влево). При этом в кольцо сдвига может входить или не входить флаг переноса.

В бит флага переноса (если он используется) записывается значение старшего бита при циклическом сдвиге влево и младшего бита при циклическом сдвиге вправо. Соответственно, значение бита флага переноса будет переписываться в младший разряд при циклическом сдвиге влево и в старший разряд при циклическом сдвиге вправо.

**Арифметические**

**Для сдвига влево**

если **of=1**, то текущее значение флага **cf** и выдвигаемого слева бита операнда различны;   
если **of=0**, то текущее значение флага **cf** и выдвигаемого слева бита операнда совпадают

**Логические**

**для сдвигов вправо**

Флаг **of** всегда сбрасывается в ноль в операциях сдвига на один разряд

**Циклические**

Особенность этого сдвига в том, что он происходит с некоторой задержкой, так как очередной сдвигаемый бит оказывается на некоторое время вне операнда.

Отличие этого сдвига от **rcl(rcr)** в том, что очередной сдвигаемый бит одновременно вдвигается в операнд справа(слева) и становится значением флага **cf.**

**Проверка битов и флагов**

Команды установки и очистки битов регистра состояния процессора (то есть флагов) позволяют установить или очистить любой флаг, что бывает очень удобно. Каждому флагу обычно соответствуют две команды, одна из которых устанавливает его в единицу, а другая сбрасывает в нуль.   
Флагу переноса C (от Carry) будут соответствовать команды **CLC** (очистка) и **SEC** или **STC** (установка).

**SAHF** Запись содержимого регистра АН в регистр флагов

**LAHF** Загрузка в **АН** содержимого регистра флагов

Команда **sahf** копирует разряды 7, 6, 4, 2 и 0 регистра **АН** в регистр флагов процессора, устанавливая тем самым значения флагов **SF**, **ZF,** **AF**, **PF** и **CF** соответственно. Команда не имеет операндов.  
Команда **sahf** (совместно с командой **lahf**) дает возможность читать и изменять значения флагов процессора, в том числе флагов **SF, ZF, AF** и **PF**, которые нельзя изменить непосредственно. Однако следует иметь в виду, что команда **sahf** заполняет только младший байт регистра флагов, поэтому нельзя изменить с ее помощью состояние флага **OF**

**28 Команды обработки строк**

/\*Команды обработки строк (цепочечные команды) позволяют производить действия над блоками байтов или слов памяти. Эти блоки (или строки) могут иметь длину до 64 Кбайт и состоять из числовых или алфавитно-цифровых значений (типа символов в кодах ASCII) и быть следующего размера: 8 бит — байт; 16 бит — слово; 32 бита — двойное слово. Всего в системе команд микропроцессора имеется семь операций-примитивов обработки цепочек. Каждая из них реализуется в микропроцессоре тремя командами, в свою очередь, каждая из этих команд работает с соответствующим размером элемента — байтом, словом или двойным словом. Особенность всех цепочечных команд в том, что они, кроме обработки текущего элемента цепочки, осуществляют еще и автоматическое продвижение к следующему. Предполагается, что строка-приемник находится в дополнительном сегменте ES, а строка-источник – в сегменте данных DS. Процессор адресует строку-приемник через регистр EDI/DI, а строку-источник – через регистр ESI/SI. Перед выполнением команд нyжно обеспечить, чтобы эти регистры содержали смещение первых обрабат. элементов обеих строк относительно начала соответствующего сегмента. **=> LEA регистр, память**

Если обе строки находятся в сегменте данных, то необходимо в регистр ES поместить адрес начала сегмента данных, т.е. значение, находящееся в регистре DS.

Если флаг DF равен 0, то значения регистров ESI/EI и EDI/DI *увеличиваются* после исполнения каждой команды. Если флаг DF равен 1, то они *уменьшаются.* Состоянием флага DF можно управлять с помощью двух команд: CLD – сбросить флаг направления, которая полагает его равным нулю, STD – установить флаг направления, которая присваивает ему значение 1.\*/

1. **Префиксы повторения**

 Нужны , чтобы одна команда обработки строк обработала группу последовательных элементов памяти. Для этого перед ней надо указать ***префикс повторения* REP.** Он представляет собой не команду, а однобайтовый модификатор, который заставляет микропро­цессор 8088 выполнить аппаратные повторения команды обработки строк. Это значительно сокращает время на обработку длинных строк по сравнению с прог­раммно-организованными циклами. Число повторений извлекается из регистра СХ. После очередного выполнения команды значение в регистре СХ уменьшается на единицу. Отличия префиксов в том, на каком основании принимается решение о циклическом выполнении цепочечной команды: **по состоянию регистра ecx/cx или по флагу нуля zf.**

Префикс повторения rep (REPeat).Этот префикс используется с командами, реализующими операции-примитивы пересылки и сохранения элементов цепочек — соответственно, **movs и stos, (ins и outs).** Префикс **rep** заставляет данные команды выполняться, пока содержимое в ecx/cx не станет равным 0. При этом цепочечная команда, перед которой стоит префикс, автоматически **уменьшает** содержимое ecx/cx на единицу.

Действия rep:   
1. анализ содержимого cx:   
 если cx<>0, то выполнить цепочечную команду, следующую за данным префиксом и перейти к шагу 2;   
 если cx=0, то передать управление команде, следующей за данной цепочечной командой (выйти из цикла по rep);   
2.уменьшить значение cx=cx–1 и вернуться к шагу 1;

**Префиксы повторения repe или repz**

**(REPeat while Equal or Zero)**

Они заставляют цепочечную команду выполняться до тех пор, пока содержимое ecx/cx не равно нулю или флаг ZF равен 1. Наиболее эффективно эти префиксы можно использовать с командами **cmps и scas** для поиска отличающихся элементов цепочек.

Действия repe и repz:   
1. анализ содержимого cx и флага zf:   
если cx<>0 или zf<>0, то выполнить цепочечную команду, следующую за данным префиксом, и перейти к шагу 2;   
если cx=0 или zf=0, то передать управление команде, следующей за данной цепочечной командой (выйти из цикла по rep);   
2. уменьшить значение cx=cx-1 и вернуться к шагу 1;

**Префиксы повторения repne или repnz**

**(REPeat while Not Equal or Zero)**

Префиксы repne/repnz заставляют цепочечную команду циклически выполняться до тех пор, пока содержимое ecx/cx не равно нулю или флаг zf равен нулю.

Данные префиксы также можно использовать с командами **cmps и scas**, но для поиска совпадающих элементов цепочек.

Действия repne и repnz:   
1. анализ содержимого cx и флага zf:   
если cx<>0 или zf=0, то выполнить цепочечную команду, следующую за данным префиксом и перейти к шагу 2;   
если cx=0 или zf<>0, то передать управление команде, следующей за данной цепочечной командой (выйти из цикла по rep);   
2. уменьшить значение cx=cx–1 и вернуться к шагу 1.

**Команды пересылки**

Команда MOVS копирует байт или слово из одной части памяти в другую. Она имеет формат **MOVS строка\_приемник, строка\_источник**  
Строка­\_источник – строка в сегменте данных, а строка\_приемник – строка в дополнительном сегменте.

После пересылки элемента команда MOVS изменяет указатели строки-источника SI и строки-приемника DI. Если флаг DF равен 0, то микропроцессор увеличивает значения регистров SI и DI после пересылки и тем самым адресуется к следующим элементам памяти. Если флаг DF равен 1, то микропроцессор уменьшает значения регистров SI и DI после пересылки и тем самым адресуется к предыдущему элементу памяти.

**MOVSB** элементы длиной в байт **MOVSW** элементы длиной в слово **не требуют операндов**

**MOVSD** элементы длиной в двойное слово

Полные физические адреса для операндов цепочечных команд следующие:   
адрес\_источника — пара **ds:esi/si**;   
адрес\_приемника — пара **es:edi/di**.

**Команды сравнения строк**

**CMPS строка\_приемник, строка\_источник**

Команда сравнения строк CMPS сопоставляет операнд-источник с операндом-приемником и возвращает результат через флаги. Команда не изменяет значения операндов. ***Алгоритм:***

1. выполнить вычитание элементов (источник - приемник), адреса элементов предварительно должны быть загружены:

адрес источника — в пару регистров *ds*:esi/si;   
адрес назначения — в пару регистров es:edi/di;   
2. в зависимости от состояния флага df изменить значение регистров esi/si и edi/di: если df=0, то увеличить содержимое этих регистров на длину элемента последовательности; если df=1, то уменьшить содержимое этих регистров на длину элемента последовательности; в зависимости от результата вычитания установить флаги: если очередные элементы цепочек не равны, то cf=1, zf=0; если очередные элементы цепочек или цепочки в целом равны, то cf=0, zf=1;  
4. при наличии префикса выполнить определяемые им действия ( repe/repne).

Команда CMPS сравнивает операнды с помощью их вычитания, а именно *вычитается операнд-приемник из операнда-источника.* Для сравнения нескольких элементов команду CMPS надо использовать с префиксами повторения REPE (REPZ) – пока CX не станет равен нулю или не будет найдена пара несовпадающих элементов; или REPNE (REPNZ) – пока СХ не станет равным нулю, либо не будет найдена пара совпадающих элементов.

**CMPSВ   
CMPSW не требуют операндов  
CMPSD**

**Команды сканирования**

Команды сканирования строк позволяют осуществить поиск заданного значения в строке. При сканировании строки байтов искомое значение должно находиться в регистре AL, а при сканировании строки слов – в регистре АХ, а при сканировании строки двойных слов – в регистре ЕАХ.

**SCAS строка\_приемник**

**SCASB   
SCASW адрес сканируемой строки ESI/SI**

**SCASD**

***Алгоритм работы:***   
1. выполнить вычитание (элемент цепочки-(eax/ax/al)). Элемент цепочки локализуется парой es:edi/di. Замена сегмента es не допускается;   
2. по результату вычитания установить флаги;   
3. изменить значение регистра edi/di на величину, равную длине элемента цепочки. Знак этой величины зависит от состояния флага df:   
df=0 — величина положительная, то есть просмотр от начала цепочки к ее концу;   
df=1 — величина отрицательная, то есть просмотр от конца цепочки к ее началу

**Команды загрузки**

Данные команды служат для того, чтобы загрузить элемент строки в регистр, имеют формат:

**LODS строка\_источник  
 lodsb    
 lodsw    
 lodsd**

*Алгоритм работы:*   
1. загрузить элемент из ячейки памяти, адресуемой парой ds:esi/si, в регистр al/ax/eax.   
2. Размер элемента определяется неявно (для команды lods) или явно в соответствии с применяемой командой ( lodsb, lodsw, lodsd);   
3. изменить значение регистра si на величину, равную длине элемента цепочки.   
4.Знак величины зависит от состояния флага df:   
df=0 — значение положительное, то есть просмотр от начала цепочки к ее концу;   
df=1 — значение отрицательное, то есть просмотр от конца цепочки к ее началу.   
Команды не влияет на флаги

**Команда записи**

Данные команды служат для того, чтобы загрузить элемент строки в регистр или изменить его, имеют формат:

**STOS строка\_приемник  
STOSB  
STOSW  
STOSD**

*Алгоритм работы:*   
1. записать элемент из регистра al/ax/eax в ячейку памяти, адресуемую парой es:di/edi.   
2. Размер элемента определяется неявно (для команды stos) или конкретной применяемой командой (для команд stosb, stosw, stosd);   
3. изменить значение регистра di на величину, равную длине элемента цепочки.   
3. Знак изменения зависит от флага df:   
df=0 — увеличить, что означает просмотр от начала цепочки к ее концу;   
df=1 — уменьшить, что означает просмотр от конца цепочки к ее началу.   
выполнение команды не влияет на флаги

***получение элементов цепочки из порта ввода-вывода:***ins адрес\_приемника,номер\_порта   
insb   
insw   
insd

***вывод элементов цепочки в порт ввода-вывода:*** outs номер\_порта,адрес\_источника   
outsb   
outsw   
outsd

**29.1. Регистры сопроцессора**

Арифметический сопроцессор содержит 8 численных 80-битовых регистров. Регистры общего назначения (стек): ST(0), ST(1), … , ST(7).

Управляющие регистры: CWR, SWR, TWR, FIP, FDR

Регистр тегов (Tags Word Register). Поля регистра тегов классифицируют содержимое соответствующего численного регистра.

Регистр управления (Control Word Register)

Регистр состояния (Status Word Register) содержит флажки особых случаев.

**29.2. Команды пересылки данных**

Запись в стек

Команды FLD, FILD, FBLD загружают в вершину стека ST(0) вещественное, целое и десятичное числа соответственно

Извлечение из стека

FSTP память <-ST(0), вещественный формат

FISTP память <-ST(0), целочисленный формат

FBSTP память <-ST(0), десятичный формат

После записи содержимое поля ST увеличивается на 1

Копирование данных: содержимое стека не изменяется

Загрузка констант

fldz 0

fld 1

fld pi

fldl2t log2(10)

fldl2e log2(e)

fldlg2 lg2

fldln2 ln2

Команда обмена fxch меняет содержимое регистра, указанного в этой команде с регистром ST(0)

**29.3. Арифметические команды**

Сопроцессор использует 6 основных типов арифметических команд, имеющих вид FXXX (FADD, FSUB, FSUBR, FMUL, FDIV, FDIVR)

Команды вида FXXX ?… : первый операнд берется из верхушки стека (источник), второй – следующий элемент стека. Результат выполнения команды записывается в стек.

Команды вида FXXX память: источник берется из памяти, приемником является верхушка стека ST(0)

Указатель стека не меняется, команда действительна только для операндов с одинарной и двойной точностью.

FIXXX память – аналогично предыдущему операнду, но операции могут быть 16- или 32-битовые целые числа

FXXX ST, ST(i): регистр ST(i) является источником, а ST(0) – приемником. Указатель стека не изменяется.

FXXX ST(i), ST: регистр ST(0) является источником, а ST(i) – приемником. Указатель стека не изменяется.

Кроме основных арифметических команд существуют еще дополнительные команды: FSQRT (квадратный корень), FSCALE (масштабирование на степень числа 2), FPREM, FRNDINT (округление до целого), FXTRACT, FABS, FCHS

**29.4. Команды сравнения чисел**

FCOM сравнение

FICOM целочисленное сравнение

FCOMP сравнение и извлечение из стека

FICOMP целочисленное сравнение и извлечение из стека

FCOMPP сравнение и двойное извлечение из стека

FTST сравнение операнда с нулем

FXAM анализ операнда

FCOM вычитает содержимое операнда, размещенного в ST(0)

С помощью команды «FSTSW AX» программа может переписать содержимое регистра состояния сопроцессора в регистр AX центрального процессора.

**29.5. Трансцендентные команды**

-тригонометрические:

Fsin sin(st)->st

fsincos sin(st)->st, cos(st)->st

fcos cos(st)

fptan tg(st)->st, fld1

fpatan arctg(st(st(1))->st

-степенные и логарифмеческие

f2xm1 2^st-1->st, -1<x<1

fy2xpl st(1)\*log2(st+1)->st(1), pop

fy12x st(1)\*log2(st)->st(1), pop

**29.6. Управляющие команды**

FINIT – очищает все регистры

FSAVE – сохраняет состояние регистров в памяти

FSTST

Синхронизация СP и FPU: wait/fwait

**30.1. Команда CALL**

Команда **call** передает управление подпрограмме, сохранив перед этим в стеке смещение к точке возврата.   
Команда **ret,** которой обычно заканчивается подпрограмма, забирает из стека адрес возврата и возвращает управление на команду, следующую за командой call. Команда не воздействует на флаги процессора.

Команда **call** имеет четыре модификации:  
- вызов прямой ближний (в пределах текущего программного сегмента);  
- вызов прямой дальний (вызов подпрограммы, расположенной в другом программном сегменте);  
- вызов косвенный ближний;  
- вызов косвенный дальний.

Все разновидности вызовов имеют одну и ту же мнемонику call, хотя и различающи-еся коды операций. Во многих случаях транслятор может определить вид вызова по контексту, если это невозможно, следует использовать атрибутные операторы:  
 **near ptr - прямой ближний вызов;** **far ptr - прямой дальний вызов;  
 word ptr - косвенный ближний вызов dword ptr - косвенный дальний вызов.**

**near ptr - прямой ближний вызов;**

Команда call прямого ближнего вызова

заносит в стек относительный адрес точки

возврата в текущем программном сегменте и

модифицирует IP так, чтобы в нем

содержатся относительный адрес точки перехода в том же программном сегменте.

**Косвенные ближние вызовы**  
 Адрес подпрограммы содержится либо в ячейке памяти, либо в регистре. Это позволя-ет, как и в случае ближнего перехода, моди-фицировать адрес вызова, а также осущест-влять вызов не с помощью метки, а по известному абсолютному адресу.

**Прямой дальний вызов far ptr**Этот вызов позволяет обратиться к подпрограмме из другого сегмента.

**Косвенный дальний вызов.**   
 Отличается от косвенного ближнего вызова лишь тем, что подпрограмма находится в другом сегменте, а в ячейке памяти содержится полный адрес подпрограммы, включающий сегмент и смещение.

Если процедура имя имеет атрибут NEAR, то команда CALL помещает смеще-ние адреса следующей команды в стек. Если процедура имя имеет атрибут FAR, то команда CALL помещает в стек содержи-мое регистра CS, а затем смещение адреса.  
 После сохранения адреса возврата команда CALL загружает смещение адреса метки имя в указатель команд IP. Если процедура имеет атрибут FAR, то команда CALL загружает также номер блока метки имя в регистр CS.

Пусть регистр ВХ содержит смещение адреса подпрограммы относительно регистра сегмента CS.  
 При исполнении этой команды микропроцессор копирует содержимое регистра ВХ в указатель команд IP, затем передает управление команде, адресуемой парой регистров CS : IP.

Процедуру с атрибутом NEAR можно вызвать косвенно, например:  
**CALL WORD PTR [ ВХ ]** ; косвенная регистровая  
**CALL WORD PTR [ BX ] [ SI ]** ; по базе с индексированием  
**CALL WORD PTR VARIABLE\_NAМE**CALL WORD PTR VARIABLE\_NAME [ BX ] ; прямая с индексированием  
**CALL MEM\_WORD   
CALL WORD PTR ES : [ BX ] [ SI ]  
 ; по базе с индексированием**

Процедуру с атрибутом FAR можно вызвать косвенно, используя переменную размером в двойное слово, например:  
**CALL DWORD PTR [ BX ]   
CALL MEM\_DWORD   
CALL DWORD PTR SS : VARIABLE\_NAME [ SI ]** Первые две команды CALL извлекают адреса подпрограмм из сегмента данных, а последняя – из сегмента стека. Подпрограмма может сама вызывать другие подпрограммы. Таким образом возможно использование вложенных процедур.  
 Т. к. каждая команда CALL помещает в стек 2 или 4 байта адреса, то число уровней вложения ограничено только размером сегмента стека.

**23.3. Команда возврата RET**(RETurn from procedure)   
RET Возврат из процедуры   
RETN Возврат из ближней процедуры   
RETF Возврат из дальней процедуры

Команда ret извлекает из стека адрес возврата и передает управление назад в программу, первоначально вызвавшую подпрограмму.  
 Если командой ret завершается ближняя процедура, (near), или исполь-зуется **retn,** со стека снимается одно слово- относительный адрес точки возврата. Передача управления в этом случае осуществляется в пределах одного программного сегмента.

Если командой **ret** завершается дальняя процедура, объявленная с атрибутом far, или используется модификация команды retf, со стека снимаются два слова: смещение и сегментный адрес точки возврата.  
 Выполнение команды не влияет на флаги.

*Применение:* Команду ret необходимо применять для возврата управления вызывающей программе из подпрограммы, управление которой было передано по команде **call.** Микропроцессор имеет три варианта команды возврата **ret - это ret, ее синоним retn, а также команда retf.**  
 Они отличаются типами процедур, в которых используются. Команды ret и retn служат для возврата из процедур ближнего типа. Команда retf — команда возврата для процедур дальнего типа. Какая конкретно команда будет использоваться, определяется компилятором; программисту лучше использовать команду ret и доверить транслятору самому сгенерировать ее ближний или дальний вариант. Количество команд ret в процедуре должно соответствовать количеству точек выхода из подпрограммы

**23.4. Способы передачи параметров.**  
**void \_cdecl: Func (int A,int B, int C, int D**) {}  
 **\_cdecl** - соглашение о вызовах (определение порядка размещения в стеке и извлечения из стека параметров, передаваемых при вызове функций   
- **прямой порядок расположения данных в стеке  
- в стек заносится сначала последний параметр, затем предпоследний и т.д.**

**- допускает переменное число параметров, поскольку стек очищает пользователь**

**void \_stdcall** **Func (int A,int B, int C, int D**){}  
-прямой порядок данных в стеке  
- в стек заносится сначала последний параметр  
- стек очищает функция поэтому число параметров фиксировано

**void \_fastcall** **Func (int A,int B, int C, int D**){}- аналогично stdcall, но два первых параметра передаются через регистры **eax** **edx**

**Существует  \_pascal:  
- обратный порядок расположения данных в стеке  
- в стек заносится сначала первый параметр  
- поэтому число параметров фиксировано**

**23.5. Правила передачи параметров**  
Использование регистров в 16-ти битном режиме или Windows, C или C++: -16-ти битное значение возвращается в AX,

-32-х битное значение в DX:AX,   
-значение с плавающей запятой в ST(0)

Регистры AX, BX, CX, DX, ES и арифметические флаги могут быть изменены подпрограммой; другие регистры должны быть сохранены и восстановлены.   
Подпрограмма может полагаться на то, что при вызове другой процедуры значение ESI, EDI, EBP, DS и SS не изменится

Использование регистров в 32-х битных Windows, C++ и других языках программирования:  
 -целочисленное значение возвращается в EAX,  
 -значение с плавающей точкой возвращается вST(0).   
Регистры EAX, ECX, EDX (но не EBX) могут быть изменены процедурой;

Bсе другие регистры должны быть сохранены и восстановлены.  
 Сегментные регистры нельзя изменять даже временно.  
 CS, DS, ES и SS указывают на плоский сегмент. FS используется операционной системой.   
GS не используется, но зарезервирован.

Флаги могут меняться процедурой, но со следующими ограничениями: флаг направления равен 0 по умолчанию. Флаг прерывания не может быть очищен. Стековый регистр плавающей запятой пуст при входе в процедуру и должен быть пустым при выходе, если только ST(0) не используется для возвращения значения. Процедура может полагаться на неизменность EBX, ESI, EDI, EBP и всех сегментных регистров при вызове другой процедуры.

**23.6. Внешние подпрограммы**

**#include <stdio.h>  
#include <stdlib.h>  
extern "C" int \_cdecl SubInt(int, int);  
extern "C" int \_cdecl SumAInt(int[], int,int\*);**

void Rnd\_array (int ar[],int \*n)  
**{// ввод размерности и элементов массива** int i,j;  
 printf("Enter dimension n \n");  
 scanf("%d",n);  
 for (i=0;i<\*n;i++)   
 ar[i]=rand()%100; ; }

void Print\_array ( int ar[],int n)  
 **// вывод элементов массива**{ printf("Items array\n");  
 for ( int i=0;i<n;i++)  
 printf("%5d",ar[i]);  
 printf("\n");}

void main()  
{int x,y;  
printf ("Enter 2 number");  
scanf("%d%d",&x,&y);  
int r=SubInt(x, y);  
// **в стеке сначала y потом x**

printf ("difference=%d\n",r);

int b[20],n,k;  
Rnd\_array(b, &n);  
Print\_array(b, n);  
SumAInt(b,n,&k);  
**// в стеке заносится сначала &k,n,&b**printf ("summa=%d\n",k); }

Подпрограмма во внешнем файле .asm  
.486  
.model flat ;**плоская модель памяти**   
.data ;описание данных  
.code

**Public \_SubInt**\_SubInt proc  
; **В стеке 4 байта адрес точки возврата**  
 push ebp ;4 байта  
 mov ebp,esp  
 mov eax,[ebp+8] ; x  
 sub eax,[ebp+12]; y   
 pop ebp  
 ret   
 SubInt endp

**Public \_SumAInt**\_**SumAInt proc  
 push ebp  
 mov ebp,esp  
 mov esi,[ebp+8] ;&array   
 mov ecx,[ebp+12];n  
 xor eax,eax**

**for1: add eax,[esi]  
 add esi,4  
 loop for1  
 mov edi,[ebp+16];&k  
 mov [edi],eax  
 pop ebp  
 ret   
\_SumAInt endp  
 end**

**23.7.Компилятор MASM**

В начале 1990-х, альтернативные ассемблеры, типа Borland TASM и условно-бесплатного ассемблера x86 NASM начали брать часть доли на рынке MASM.   
 Позже в 2000 году, MASM 6.15 был выпущен как часть Visual C++ пакета разработки, который является бесплатным.  
 В результате все версии Visual C++ позже чем 6.0, включали версию MASM, равную версии Visual C++.

Позже в Visual C++ 2005 появилась 64-битовая версия MASM (название линкера — ml64.exe).   
 Вместе с большим сообществом программистов MASM, эти события помогли остановить снижение популярности MASM до других ассемблеров.   
 Сегодня MASM — все ещё ассемблер номер один на платформе Win32, несмотря на конкуренцию с новыми продуктами, как NASM, FASM и HLA.

**К**омпилятор MASM создан специально для написания программ на ассемблере для Win32. В нём есть макросы и специальные директивы для упрощения программирования.   
**Функции.** Основное преимущество MASM это макрос **invoke,** он позволяет вызывать API функции по-обычному с проверкой количества и типа параметров. Это почти тот же call, как в TASM, но этот макрос проверяет количество параметров и их типы. Вот так вызывается функция**:   
     Invoke <функция>, <параметр1>, <параметр2>, <параметр3>**