Лекция 4

Указатели, Функции

Типы данных, определяемые пользователем

В реальных задачах информация, которую требуется обрабатывать, может иметь достаточно сложную структуру. Для ее адекватного представления используются типы данных, построенные на основе простых типов данных, массивов и указателей.

Язык С++ позволяет программисту определять свои типы данных и правила работы с ними.

Переименование типов (typedef)

Для того чтобы сделать программу более ясной, можно задать типу новое имя с помощью ключевого слова **typedef**:

typedef тип новое_имя [размерность];

Размерность может отсутствовать.

Примеры:

```
typedef unsigned int UINT;
typedef char Msg[100];
typedef struct { char fio[30]; int date, code; float salary;
                 } Worker;
```

Введенное таким образом имя можно использовать таким же образом, как и имена стандартных типов:

```
UINT   i, j;        // две переменных типа unsigned int
Msg  str[10];       // массив из 10 строк по 100 символов
```

Перечисления (enum)

При написании программ часто возникает потребность определить несколько именованных констант, для которых требуется, чтобы все они имели различные значения. Для этого удобно воспользоваться перечисляемым типом данных. Формат:

enum [имя_типа] { список_констант };

Имя типа задается в том случае, если в программе требуется определять переменные этого типа. Компилятор обеспечивает, чтобы эти переменные принимали значения только из списка констант.

Константы должны быть целочисленными и могут инициализироваться обычным образом. При отсутствии инициализатора первая константа обнуляется, а каждой следующей присваивается на 1 большее значение, чем предыдущей:

```
enum Err {ERR_READ, ERR_WRITE, ERR_CONVERT};
 Err error;
 // ...
 switch (error) {
case  ERR_READ:    /* операторы */ break;
case  ERR_WRITE:  /* операторы */ break;
case ERR_CONVERT:   /* операторы */ break;
                    }
```

Константам ERR_READ, ERR_WRITE, ERR_CONVERT присваиваются значения 0, 1 и 2 соответственно.

Структуры (struct)

В отличие от массива, все элементы которого однотипны, структура может содержать элементы разных типов. Во многих случаях достаточно использовать структуры так, как они определены в языке С:

```
struct [ имя_типа ]  {
 тип_1 элемент_1;
 тип_2 элемент_2;
 // ...
тип_n элемент_n;   }
      [ список_описателей ];
```

Элементы структуры называются *полями структуры* и могут иметь любой тип, кроме типа этой же структуры, но могут быть указателями на него.

Если отсутствует имя типа, должен быть указан список описателей переменных, указателей или массивов. В этом случае описание структуры служит определением элементов списка:

```
struct {
 char fio[30];
 int date, code;
 float salary;
       } stuff[100], *ps;   // определение массива структур и указателя на структуру
```

Если список отсутствует, описание структуры определяет новый тип, имя которого можно использовать в дальнейшем наряду со стандартными типами, например:

```
struct  Worker  {   // описание нового типа Worker
 char  fio[30];
 int  date,  code;
float  salary;
};              // описание заканчивается точкой с запятой
Worker stuff[100], *ps;        // определение массива типа Worker и указателя на тип Worker
```

Инициализация структуры

Для *инициализации структуры* значения ее элементов перечисляют в фигурных скобках в порядке их описания:

```
Struct   {
 char fio[30];
 int date, code;
 float salary;
          } worker = {"Страусенко", 31, 215, 3400.55};
```

Для переменных одного и того же структурного типа определена *операция присваивания,* при этом происходит поэлементное копирование.

Структуру можно передавать в функцию и возвращать в качестве значения функции. Другие операции со структурами могут быть определены пользователем.

Доступ к полям структуры выполняется с помощью операций выбора . (точка) при обращении к полю через имя структуры и -> при обращении через указатель, например:

```
Worker worker, stuff[100], *ps;
 // ...
worker.fio = "Страусенко";
 stuff[8].code = 215;
 ps->salary = 0.12;
```

Указатели.

Зачем нужны указатели?

Применение указателей позволяет упростить алгоритм или повысить его эффективность. Каким образом? Указатели могут обесечить простые способы ссылок на массивы, списки или блоки данных. Для таких ссылок достаточно иметь простой элемент данных: указатель. Нередко бывает проще и эффективнее манипулировать простым указателем, чем управлять полным списком данных. Управление памятью компьютера - это еще одно из важнейших применений указателей.

Указатель — переменная, диапазон значений которой состоит из адресов ячеек памяти или специального значения — ***нулевого адреса*** *(*для указания того, что в данный момент там ничего не записано*).*

В C++ указатель объявляется с помощью звездочки:

Int *x; //переменная **x** есть указатель на **int**

Чтобы указатель обрабатывал значение, а не адрес памяти, используется операция разыменования:

Printf(“%d”,*x); //звездочка - это оператор разыменования

***x = 100;**

В C++ у любой переменной, в том числе и не указателя можно узнать адрес памяти, по которому она расположена. Делается это с помощью оператора **&:**

x=&a;

Пример. Связь указателя с переменной

```
#include <conio.h>
#include <iostream.h>
void main()
{
  clrscr();
    int *ptr;      //ptr есть указатель на int
    int a=100;    //a является обычной переменной
   printf (“*ptr =%d”, *ptr);    //Чтобы получить значение ptr -
     //разыменовали указатель ptr .
                  // При этом  если указатель не был связан
                   // ни с какими значениями, можно получить что угодно.
   ptr = &a;   // Взяли адрес у переменной a и присвоили этот адрес
     // в   указатель
 printf (“*ptr =%d”, *ptr); // Указатель теперь указывает на a.
                                      // Значит *ptr = a = 100
getch();
return;
}
```

Двойственная природа указателя.

Важно:

Если указатель указывает на некоторый один адрес памяти, то и работает он со значением из этого адреса.

а) Изменяя значение переменной по адресу на который указатель указывает – изменится и значение разыменовываемого указателя

б) При присвоении значений разыменованному указателю – изменится значение переменной по указываемому указателем адресу

Вариант а: Изменение значения переменной для указателя

```
#include <conio.h>
#include <iostream.h>
void main()
{
clrscr();
   int a=100;
   int *ptr;   // ptr есть указатель на int
   ptr=&a;    // Указатель ptr = адрес переменной a
  printf (“a = %d”, *ptr );      // т.к. указатель опирается на адрес переменной a,
                                  // любое изменение a влияет на то, что отображает
                                    //разыменованный указатель:   *ptr=a=100
   a=999;   //поменяли значение переменной
 printf (“a = %d”, *ptr );    //указатель опирается на адрес переменной a:
                                    //*ptr=a=999

getch();
return;
}
```

Вариант б. Изменение разыменованного указателя влияет на переменную по адресу указателя

```
#include <conio.h>
#include <iostream.h>
void main()
{
clrscr();
   int a=100;    //Разные переменные
   int *ptr;     //ptr есть указатель на int
   int c=55;
  ptr=&a;    // Указатель = адрес переменной a => *ptr=a=100;
*ptr=999;   // По адресу переменной a записалось новое
                // значение  a=*ptr=999;
  ptr=&c;   // Указатель теперь указывает на адрес c.
               // Влияние указателя на a прекращено
  *ptr=88;   //c изменилось было 55 - стало 88
 printf ("a = %d", a);
 printf ("c = %d", c);
getch();
return;
}
```

Указатели на указатели

Указатели могут ссылаться на другие указатели. При этом в ячейках памяти, на которые будут ссылаться первые указатели, будут содержаться не значения, а адреса вторых указателей. Число символов * при объявлении указателя показывает порядок указателя. Чтобы получить доступ к значению, на которое ссылается указатель его необходимо разыменовывать соответствующее количество раз.

int var = 123; // инициализация переменной var числом 123

int *ptrvar = &var; // указатель на переменную var

int **ptr_ptrvar = &ptrvar; // указатель на указатель на
// переменную var

int ***ptr_ptr_ptrvar = &ptr_ptrvar; // указатель на
// указатель на указатель на переменную var

Обычно, указатели порядка выше первого используются редко.

Операции над указателями в C++

Над указателями определено 5 основных операций.

- **Определение адреса указателя: &p**, где p – указатель (&p – адрес ячейки, в которой находится указатель).
- **Присваивание.** Указателю можно присвоить адрес переменной p=&q, где p – указатель, q – идентификатор переменной.
- **Определение значения, на которое ссылается указатель: *p** (операция косвенной адресации).
- **Увеличение (уменьшение) указателя.** Увеличение выполняется как с помощью операции сложения (+), так и с помощью операции инкремента (++). Уменьшение – с помощью операции вычитания (–) либо декремента (—).

  Например, пусть p1 – указатель, тогда p1++ перемещает указатель на:
  - 4 байта, если *p1 имеет тип int (в 32 разр-й операц-й системе) или 2 байта (в 16 разр-й операц-й системе);
  - 1 байт, если *p1 имеет тип char;
  - 4 байта, если *p1 имеет тип float.
- **Разность двух указателей.** Пусть p1 и p2 – указатели одного и того же типа. Можно определить разность p1 и p2, чтобы найти, на каком расстоянии друг от друга находятся элементы массива.

Пример программы.

Даны адреса переменных **&a=63384, &b=64390, &c=64404**. ***Что напечатает ЭВМ?***

```
# include <stdio.h>
int main()
{
float a,*p1;
int b,*p2;
char c,*p3;
a=2.5; b=3; c='A';
p1=&a; p2=&b; p3=&c;
p1++; p2++; p3++;
printf("\n p1=%p, p2=%p, p3=%p",p1,p2,p3);
return 0;
}
```

Ответ: p1=63388, p2=64392, p3=64405.

Операции адресной арифметики

Операции адресной арифметики подчиняются следующим правилам:

- После **увеличения значения переменной-указателя на 1** данный указатель будет ссылаться на следующий объект своего базового типа. После **уменьшения** – на предыдущий объект. Для всех указателей адрес увеличивается или уменьшается на величину, равную размеру объекта того типа, на который они указывают. Поэтому указатель всегда ссылается на объект с типом, тождественным базовому типу указателя.
- Применительно к указателям на объект типа char операции адресной арифметики выполняются как обычные арифметические операции, потому что длина объекта char всегда равна 1.
- Операции адресной арифметики не ограничены увеличением (инкрементом) и уменьшением (декрементом). К указателям, например, можно **добавлять или вычитать из них константу**. При этом значение указателя изменяется на величину этой константы, умноженной на размер объекта данного типа sizeof(тип).
- При операции **вычитания двух указателей** можно определить количество объектов, расположенных между адресами, на которые указывают эти два указателя. При этом необходимо, чтобы указатели имели один и тот же тип.
- Кроме того, стандартом C допускается **сравнение двух указателей**. Как правило, сравнение указателей может оказаться полезным только тогда, когда два указателя ссылаются на общий объект, например, на массив.
- Операции **суммирования двух указателей, и** все остальные операции

Приоритет операций над указателями

При записи выражений с указателями следует обращать внимание на приоритеты операций. В качестве примера рассмотрим последовательность действий, заданную в операторе

***p++ = 10;**

Операции разадресации и инкремента имеют одинаковый приоритет и выполняются справа налево, **но**, поскольку инкремент постфиксный, он выполняется после выполнения операции присваивания. Таким образом, сначала по адресу, записанному в указателе p, будет записано значение 10, а затем указатель будет увеличен на количество байт, соответствующее его типу. То же самое можно записать подробнее:

***p = 10; p++;**

Выражение **(*p)++** напротив, инкрементирует

Функции в C++

Реальные программы состоят из тысяч, десятков тысяч и, даже миллионов строк кода. Чтобы легче было управлять этим кодом его разбивают на подпрограммы. В языке C++ подпрограммы представляют собой функции. Такое разбиение позволяет быстрее отлавливать ошибки, повышает читаемость кода и имеет много преимуществ.

В C++ самой первой всегда выполняется функция **main()**, а остальные функции выполняются после.

Обычно программы пишутся объединяя множество таких функций (или модулей), которые могут быть описаны в разных заголовочных файлах. Однако, можно использовать и пользовательские функции, то есть можно создавать свои собственные. Для создания функций сначала нужно объявить прототип функции, а затем написать её реализацию.

Сначала задаётся **объявление функции** (прототип функции), в которой указывается, какой тип данных она возвращает, и какого типа данных будут параметры этой функции. При объявлении функции **в конце должна быть точка с запятой**.

Определение функции (реализация функции) описана после главной функции программы. В определении функции нужно указать тип возвращаемого значения, а также параметры функции и их типы данных.

После этого в фигурных скобках описывается сама реализация этой функции. Если функция имеет тип данных не void, то в теле функции обязательно должен присутствовать оператор return, который возвращает результат соответствующего типа данных.

Пример функции.

Пример функции, возвращающей сумму двух целых величин:

```
#include <stdio.h>
int sum(int a, int b);      // объявление функции
int main()
{ int a = 2, b = 3, c, d;
c = sum(a, b);   // вызов функции
scanf("%d", &d);
printf("%d", sum(c, d));   // вызов функции
return 0;
}
int sum(int a, int b)   // определение функции
{return (a + b); }
```

В определении, в объявлении и при вызове одной и той же функции типы и порядок следования параметров должны совпадать.

Объявление функции

В качестве прототипа функции может также служить ее определение, если оно находится в программе до первого вызова этой функции. Вот, например, правильная программа:

```
#include <stdio.h>      /*Это определение будет также служить и прототипом внутри
                                                                этой  программы.*/
void f(int a, int b)
{
printf("%d ", a % b);
 }
int main ()
 {
f(10,3);
return 0;
 }
```

В этом примере специальный прототип не требуется; так как функция f() определена еще до того, как она начинает использоваться в main(). Хотя определение функции и может служить ее прототипом в малых программах, но в больших такое встречается редко — особенно, когда используется несколько файлов.

Единственная функция, для которой не требуется прототип — это main(), так как это первая функция, вызываемая в начале работы программы.

Прототипы функций позволяют "отлавливать" ошибки еще до запуска программы. Кроме того, они запрещают вызов функций при несовпадении типов (т.е. с неподходящими аргументами) и тем самым помогают проверять правильность программы.

Определение функции

Определение функции содержит, кроме заголовка функции, *тело* функции, представляющее собой последовательность операторов и описаний в фигурных скобках:

тип имя ([список_параметров])
{ тело функции }

- Тип возвращаемого функцией значения может быть любым, кроме массива и функции (но может быть указателем на массив или функцию). Если функция не должна возвращать значение, указывается тип void.
- Список параметров определяет величины, которые требуется передать в функцию при ее вызове. Элементы списка параметров разделяются запятыми. Для каждого параметра, передаваемого в функцию, указывается его тип и имя (в объявлении имена можно опускать).

Для вызова функции в простейшем случае нужно указать ее имя, за которым в круглых скобках через запятую перечисляются имена передаваемых аргументов.

Вызов функции может находиться в любом месте программы, где по синтаксису допустимо выражение того типа, который формирует функция. Если тип возвращаемого функцией значения не void, она может входить в состав выражений или, в частном случае

Локальные и глобальные переменные

- Все величины, описанные внутри функции, а также ее параметры, являются локальными. Областью их действия является функция. При вызове функции, как и при входе в любой блок, в стеке выделяется память под локальные автоматические переменные.
- При выходе из функции соответствующий участок стека освобождается, поэтому значения локальных переменных между вызовами одной и той же функции не сохраняются.

При совместной работе функции должны обмениваться информацией. Это можно осуществить с помощью глобальных переменных, через параметры и через возвращаемое функцией значение.

- Глобальные переменные видны во всех функциях, где не описаны локальные переменные с теми же именами, поэтому использовать их для передачи данных между функциями очень легко. Тем не менее, использовать этот способ не рекомендуется, поскольку это затрудняет отладку программы и препятствует помещению функций в библиотеки общего пользования.
- Нужно стремиться к тому, чтобы функции были максимально

Возвращаемое значение

Возврат из функции в вызвавшую ее функцию реализуется оператором

return [выражение];

Функция может содержать несколько операторов return. Если функция описана как void, выражение не указывается. Выражение, указанное после return, неявно преобразуется к типу возвращаемого функцией значения и передается в точку вызова функции.

Примеры:

int f1() { return 1; } //правильно

void f2() { return 1; } //неправильно, f2 не должна возвращать значение

double f3() { return 1; } //правильно, 1 преобразуется к типу double

Важно !

Нельзя возвращать из функции указатель на локальную переменную.

Пример:

```
int* f()
{ int a = 5;
```

Параметры функции

Параметры, перечисленные в заголовке описания функции, называются ***формальными***, а записанные в операторе вызова функции — ***фактическими*** (или *аргументами*).

При вызове функции в первую очередь вычисляются выражения, стоящие на месте фактических параметров; затем в стеке выделяется память под формальные параметры функции в соответствии с их типом, и каждому из них присваивается значение соответствующего фактического параметра. При этом проверяется соответствие типов и при необходимости выполняются их преобразования. При несоответствии типов выдается диагностическое сообщение.

Существует два вида передачи величин в функцию: *по значению и по адресу.*

При передаче *по значению* в стек заносятся копии значений фактических параметров, и операторы функции работают с этими копиями. Доступа к исходным значениям параметров у функции нет, а, следовательно, нет и возможности их изменить.

При передаче *по адресу* в стек заносятся копии адресов параметров, а функция осуществляет доступ к ячейкам памяти по этим адресам и может изменить исходные значения параметров.

Параметры функции. Пример.

```
#include <stdio.h>

 void f(int i, int* j, int& k);
 int main()
 {
 int i = 1, j = 2, k = 3;
Printf(" i  j  k \n");
Printf("%d %d %d\n", i,  j, k);
f(i, &j, k);
Printf("%d %d %d\n", i,  j, k);
return 0;
 }
void f(int i, int *j, int &k)
 { i++; (*j)++; k++; }
```

Результат работы программы:

```
I  j  k
1  2  3
1  3  4
```

Первый параметр (i) передается по значению. Его изменение в функции не влияет на исходное значение.

Второй параметр (j) передается по адресу с помощью указателя, при этом для передачи в функцию адреса фактического параметра используется операция взятия адреса, а для получения его значения в функции требуется операция разыменования.

Третий параметр (k) передается по адресу с помощью ссылки.

При передаче *по ссылке* в функцию передается адрес указанного при вызове параметра, а внутри функции все обращения к параметру неявно разыменовываются. Поэтому использование ссылок вместо указателей улучшает читаемость программы. Использование ссылок вместо передачи по значению более эффективно, поскольку не требует копирования параметров.

Передача массивов в качестве параметров

Массив всегда передается по адресу. При этом информация о количестве элементов массива теряется, поэтому следует передавать его размерность через отдельный параметр.

Если размерность массива является константой, проблем не возникает, поскольку можно указать ее и при описании формального параметра, и в качестве границы циклов при обработке массива внутри функции.

```
#include <stdio.h>
int sum(const  int* mas, const int n);
 int const n = 10;

 int main()
{
int marks[n] = {3, 4, 5, 4, 4};
Printf( "Сумма элементов массива  =  %d",  sum(marks, n));
return 0;
}

 int sum(const int* mas, const int n)      /* варианты:    int sum (int mas[], int n)     или      int sum (int mas[n], int n)
                                                                                                    (  n   должна быть   константой) */
{
 int s = 0;
 for (int i = 0 ; i < n; i++)    s += mas[i];
return s;
 }
```

Первый пособ передачи Массива в

Имеется три способа объявления параметра, предназначенного для получения указателя на массив.

Во-первых, он может быть объявлен как массив, как показано ниже:

```
#include <stdio.h>
void display(int num[10]);

int main (void)              /* вывод чисел */
{
int  t [10], i;
for (i=0; i<10; ++i) t[i]=i;
display(t);
return 0;
}

void display(int num[10])
{
int i;
for (i=0; i<10; i++) printf ("%d", num[i]);
}
```

Хотя параметр **num** объявляется как целочисленный массив из десяти элементов, С автоматически преобразует его к целочисленному указателю, поскольку не существует параметра, который мог бы на самом деле принять весь массив. Передается только указатель на массив, поэтому должен быть параметр, способный принять его.

Второй пособ передачи Массива в функцию

Следующий способ состоит в объявлении параметра для указания на безразмерный массив, как показано ниже:

```
void display(int num[])
{
int i;
for (i=0; i<10; i++) printf("%d ", num[i]);
}
```

где **num** объявлен как целочисленный массив неизвестного размера. Поскольку С не предоставляет проверку границ массива, настоящий размер массива не имеет никакого отношения к параметру (но, естественно, не к программе). Данный метод объявления также определяет **num** как целочисленный указатель.

Третий пособ передачи Массива в функцию

Последний способ, которым может быть объявлен **num**, - это наиболее типичный способ, применяемый при написании профессиональных программ, - через указатель, как показано ниже:

```
void display(int *num)
{
int i;
for (i=0; i<10; i++) printf ("%d ", num[i]);
}
```

Он допустим, поскольку любой указатель может быть индексирован с использованием [ ], если он является массивом. (На самом деле массивы и указатели очень тесно связаны друг с другом.)

Все три метода объявления параметра приводят к одинаковому результату - указателю.

Передача двумерных массивов

При передаче *многомерных массивов* все размерности, если они не известны на этапе компиляции, должны передаваться в качестве параметров.

Внутри функции массив интерпретируется как одномерный, а его индекс пересчитывается в программе.

Например, надо передать в функцию 2-х мерный массив:

```
#include <stdio.h>
void print2DArray(int *m,  int i,   int j)
{ int ii, jj;
 for (ii = 0; ii < i; ++ii)
   {   for (jj = 0; jj < j; ++jj)   { printf  (  “%5d ",   m[ ii*j  +  jj ]  ); }
         printf("\n");
    }
 }
 int main()
{ int array2D[2][3]=
        {0, 1,
          2, 3,
          10, 11};
print2DArray  ( &array2D[0][0], 2, 3);
return 0;
}
```

Функции с переменным числом параметров

Переменный список параметров задается в заголовке функции многоточием. Список параметров совсем пустой быть не может, должен быть прописан хотя бы один явный параметр, адрес которого мы можем получить при выполнении программы. Заголовок такой функции может выглядеть так:
int f(int k...)

При этом типы отсутствующих параметровип должны совпадать с типом первого параметра, так как доступ к элементам списка аргументов осуществляется путём увеличения значения указателя на соответствующее значение (в нашем примере : sizeof(**double**)=8) .
Количество параметров становятся известными только при вызове функции.

Одним из простых примеров может служить функция, вычисляющая среднее арифметическое своих аргументов:

```
double f(double n, ...)    //--заголовок с переменным числом параметров
 { double *p = &n;   //--установились на начало списка параметров
double sum = 0, count = 0;
 while (*p)     //--пока аргумент не равен нулю
{ sum+=(*p);   //--суммируем аргумент
p++;         //--«перемещаемся на следующий аргумент
count++;       //--считаем количество аргументов }
 return ( (sum) ? sum/count : 0);          //--вычисляем среднее арифметическое
}
```

Вызов такой функции может выглядеть таким образом:
double y = f (1.0, 2.0, 3.0, 4.0, 0.0);

Переменная **y** получит значение **2.5** .

Семейство функций **printf** является одним из распространённых примеров функций с переменным количеством аргументов.

Функции с переменным числом параметров

Вот классический пример функции, принимающей переменное число параметров.

Функция возвращает сумму своих параметров. Обратите внимание, что первым параметром мы передаем число чисел для суммирования (т. е. сам первый параметр суммироваться не будет, он говорит только, сколько всего параметров будут суммироваться (это все оставшиеся параметры)).

```
#include <iostream.h>
 int sum(int n, ...)   {          // Задаем функцию с переменным числом параметров.
 int *p = &n;     //  Получаем адрес первого параметра.
 p++;          // Переводим указатель на второй параметр.
 int res = 0;        // Объявляем переменную для суммы и присваиваем ей ноль.
for(int i=0; i<n; i++)    // Сумирование оставшихся параметров.
{  res+=(*p);          // Добавление к сумме очередного параметра.
 p++;               }   // Первод указателя на следующий параметр.
 return res;          }          // Возврат суммы.

 void main()
{
int r = 0; // Суммируем 5 чисел.
r = sum(5, 1, 2, 3, 4, 500);
Printf ("Sum =%d\n" , r );
}
```

Переменное число параметров в функции обозначаается посредством многоточия (...). В нашем объявлении функции мы указываем, что обязотельно должен присутствовать первый параметр (типа int), после которого может быть любое число параметров любого типа.

Внутри функция устроена так - мы получаем адрес в адресном пространстве, по которому расположены передаваемые в функцию параметры. Это адрес первого параметра: **... int *p = &n;**

Далее мы перебираем все параметры (а всего их **n**) через указатель - он постоянно переводится на следующий параметр посредством строки: **... p++;**

Результатом выполнения указанного фрагмента будет 510.

Рекурсивные функции

Ситуацию, когда функция тем или иным образом вызывает саму себя, называют рекурсией. Рекурсия, когда функция обращается сама к себе непосредственно, называется прямой; в противном случае она называется косвенной.

Все функции языка C++ (кроме функции main) могут быть использованы для построения рекурсии.

В рекурсивной функции обязательно должно присутствовать хотя бы одно условие, при выполнении которого последовательность рекурсивных вызовов должна быть прекращена.

Обработка вызова рекурсивной функции в принципе ничем не отличается от вызова функции обычной: перед вызовом функции в стек помещаются её аргументы, затем адрес точки возврата, затем, уже при выполнении функции – автоматические переменные, локальные относительно этой функции. Но если при вызове обычных функций число обращений к ним невелико, то для рекурсивных функций число вызовов и, следовательно, количество данных, размещаемых в стеке, определяется глубиной рекурсии. Поэтому при рекурсии может возникнуть ситуация переполнения стека.

Если попытаться отследить по тексту программы процесс выполнения рекурсивной функции, то мы придем к такой ситуации: войдя в рекурсивную функцию, мы “движемся” по ее тексту до тех пор, пока не встретим ее вызова, после чего мы опять начнем выполнять ту же самую функцию сначала. При этом следует отметить самое важное свойство рекурсивной функции - ее первый вызов еще не закончился.

Пример1 Рекурсивной функции

Задача: Вычислить n!

Определение факториала рекурсивно: 0!=1; n!=(n-1)!*n при n=1,2,3, …

В соответствии с этим определение функции, вычисляющей факториал, можно

записать следующим образом:

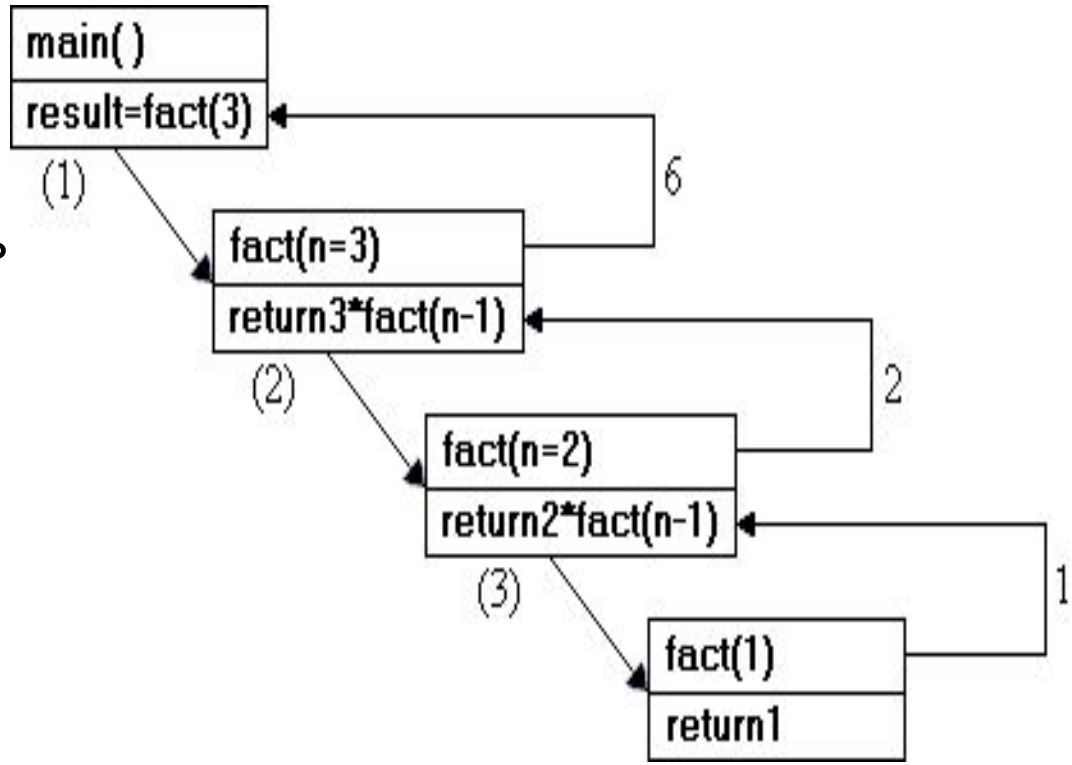
```
long fact (int n)
{   if ( n<1 ) return 1;
        else return n*fact(n-1);
}
```

Если, например, в main написать

long result=fact(3),

то последовательность

вызовов можно показать так:



Пример2 Рекурсивной функции

Задача: По заданному целому числу распечатать символьную строку цифр, изображающую это число:

```
void cnum(int n)
{   int a=10;
     if(n = = 0) return;
          else   {    cnum(n/a);   printf("%c" , n%a +'0')   }
 }
```

При косвенной рекурсии осуществляется перекрёстный вызов функциями друг друга. Хотя бы в одной из них должно быть условие, вызывающее прекращение рекурсии.

Пусть функция f1() вызывает f2(), которая, в свою очередь, обращается к f1(). Пусть первая из них определена ранее второй. Для того чтобы иметь возможность обратиться к функции f2() из f1(), мы должны поместить объявление имени f2 раньше определения обеих этих функций:

```
void f2();

     void f1()   {
                    …
                    if (…);
                      f2();
                           …}

      void f2()   {
                      …
                      f1();
                    …}
```

Пример3 Рекурсивной функции

Задача: Написать рекурсивную функцию для вычисления элемента ряда Фибоначчи с номером n. Соотношение ряда Фибоначчи задаётся формулой **f(n)=f(n-1)+f(n-2), n=0,1,..., f(0)=0, f(1)=1**

```
#include <iostream.h>
 #include <stdio.h>
 #include <stdlib.h>
long int fibo (int n)
{ if (n==0) return 0;
        else   if (n==1) return 1;
                    else return (fibo(n-1)+fibo(n-2));
}
void main ()
 { int n;
Printf( "Введите число N “);
 scanf(“%d” , &n);
 if (n<0) { printf("Число должно быть не меньше 0!“);    getchar();   exit (1);  }
long int f;
 f = fibo (n);
 printf(“Число Фибоначчи с номером %d = %l”, n ,  f);
getchar();
}
```