НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ УНИВЕРСИТЕТ «МИФИ»
Кафедра информатики и процессов управления (№17)

Дисциплина «Информатика (основы программирования)»

Методические указания

Тематическое занятие 5

Функции. Передача параметров. Рекурсия.

Содержание

Создание функций

Предназначение функций

При создании программы для решения сложной задачи выполняется разделение (*декомпозиция*) задачи на подзадачи, а подзадач – на еще меньшие фрагменты и так далее, до элементарных подзадач, которые легко программировать.

Все языки высокого уровня содержат средства для разделения программ на самостоятельные части – *подпрограммы*. В языке С подпрограммы называются ***функциями*****.** Функция позволяет скомпоновать группу операторов для выполнения единого действия, чтобы затем обращаться к ней столько раз, сколько потребуется.

Обычно программы на языке С состоят из множества мелких функций. Даже действие, которое выполняется только один раз, следует выделять в отдельную функцию, если это проясняет структуру и смысл кода программы.

Функция должна иметь уникальное имя (идентификатор), которое употребляется для ее ***вызова*****.** Такой вызов функции можно выполнять многократно. Функции можно передавать данные и она может возвращать результаты вычислений.

Функции делятся на *стандартные* и определенные программистом. Стандартные функции являются частью стандартной библиотеки С, они объявлены в соответствующих стандартных *заголовочных файлах*. Например, функции `printf` и `scanf`, объявлены в заголовочном файле `<stdio.h>`.

Функции позволяют программистам пользоваться разработками предшественников, а не начинать все заново. Если функция составлена правильно, то нет нужды знать ***как*** она работает, достаточно знать ***что*** именно она делает.

Определение функций

Определение функции имеет следующую форму:

```
ТипВозвращЗнач ИмяФункции(ОбъявленияПараметров)
{
    /* объявления и операторы (тело функции) */
}
```

Например, составим свою функцию для возведения числа ***a*** в целую положительную степень ***n***. Наша функция `power`, будет возвращать значение a^n, а числа `a` и `n` будем передавать в нее в качестве параметров `power(a,n)`:

```
int power(int a, int n) /* заголовок функции power */
{
     int i, p=1; /* локальные переменные */
     for (i=1; i<=n; ++i)
          p*=a;
     return p; /* возвращаемое значение */
}
```

Чтобы использовать эту функцию, ее нужно поместить в программу и вызвать из главной функции `main()`.

Функции и структура программы

В языке С нельзя определить функцию внутри другой функции. Программа на С состоит из функций, которые являются внешними по отношению друг к другу. При этом каждая функция может вызывать каждую другую.

Определения функций могут следовать в произвольном порядке, а также находиться в нескольких файлах исходного кода программы. Пока будем полагать, что все функции находятся в одном файле.

Главная функция программы `main()` подчиняется тем же правилам. Но при запуске программы выполняется только функция `main()`, а остальные функции могут быть вызваны из нее напрямую или посредством других функций.

Составим программу, которая вызывает нашу функцию `power(a,n)` для вычисления значения x^y :

```
#include <stdio.h>

int power(int a, int n); /* прототип функции power */
int main(void) /* главная функция программы */
{
     int x, y, res; /* локальные переменные */
     printf("Input x: "); scanf("%d", &x);
     printf("Input y: "); scanf("%d", &y);
     res = power(x,y); /* вызов функции power */
     printf("Result x^y=%d\n", res);
     return 0;
}
int power(int a, int n) /* заголовок функции power */
{
     int i, p=1; /* локальные переменные */
     for (i=1; i<=n; ++i)
          p*=a;
     return p; /* возвращаемое значение */
}
```

Прототипы функций

В данном случае определение функции `power` следует после главной функцией `main()`, поэтому до функции `main()` необходимо описать ***прототип*** функции `power` – следующую строку:

```
int power(int a, int n);
```

Прототип сообщает, что `power` является функцией с двумя параметрами типа `int`, возвращающей значение тоже типа `int`.

Прототип функции обязан согласовываться по форме как с заголовком в определении функции, так и с каждым ее вызовом. В противном случае возникает ошибка.

При этом имена параметров согласовывать не обязательно, можно даже не указывать их. Поэтому прототип функции `power` можно записать так:

```
int power(int, int);
```

Теперь переработаем функцию `main()`, чтобы продемонстрировать многократный вызов функции `power`:

```
#include <stdio.h>
int power(int, int); /* прототип функции power */
int main(void) {
     int x, y;
     printf("Calculate x^y (input x=0,y=0 for finish).\n");
     do {
          printf("Input x: "); scanf("%d", &x);
          printf("Input y: "); scanf("%d", &y);
          if (y>=0 && (x||y)) {
               printf("Result x^y=%d\n", power(x,y));
          } else
               printf("Error: Incorrect value.\n");
     } while (x||y);
     return 0;
}
int power(int a, int n) { /* заголовок функции power */
     int i, p=1; /* локальные переменные */
     for (i=1; i<=n; ++i)
          p*=a;
     return p; /* возвращаемое значение */
}
```

Здесь функция `power` вызывается непосредственно как один из параметров стандартной функции `printf`.

Передача параметров

Параметры функций и локальные переменные

Параметры, объявленные в заголовке функции при ее определении, называют ***формальными параметрами***. При объявлении параметры указывают через запятую, каждый со своим типом.

Переменные, массивы и другие данные, описанные внутри функции, называются ***локальными***.

В рассмотренном примере `a` и `n` – формальные параметры функции `power`, а `i` и `p` – локальные переменные. И те, и другие доступны только в пределах этой функции и не доступны из вызывающей функции.

Вызов функций

Вызов функций осуществляется указанием ее имени и списка ***фактических параметров*** через запятую.

Фактические параметры – это параметры, которые передаются функции при ее вызове. Фактические параметры так же называют ***аргументами*** функции.

Количество, тип и порядок следования формальных и фактических параметров должны совпадать.

При вызове функции `power` работа главной функции `main()` приостанавливается, и управление вычислительным процессом передается на участок кода, занимаемый функцией `power`. После окончания выполнения функции `power`, управление возвращается в точку вызова, и работа функции `main()` продолжается.

Оператор `return`

Оператор `return` – это механизм возвращения значения из вызываемой функции в вызывающую. После `return` может идти любое выражение, которое преобразуется к типу, возвращаемому функцией согласно ее определению.

```
return Выражение;
```

Если после `return` выражение не указано, то в вызывающую функцию ничего не передается.

После выполнения оператора `return` выполнение функции прекращается, и управление передается в точку ее вызова. То же самое происходит. если достигнут конец тела функции.

Отсутствие параметров

Если в процедуру или функцию не передаются данные, то в ее заголовке необходимо указать отсутствие значения с помощью типа `void`. Например:

```
int myfunc(void) {
     ...
     return ...
}
```

При вызове такой функции скобки опускать нельзя, но их нужно оставлять пустыми

```
f = myfunc();
```

Тип `void` также следует указывать, если функция не возвращает никаких значений. Пример определения минимальной функции-заглушки:

```
void myfunc(void) {}
```

и вызова этой функции

```
myfunc();
```

Способы передачи параметров

Передача по значению

В языке С все параметры функций передаются ***«по значению»***. Это означает, что вызываемая функция получает значения своих параметров в виде временных копий передаваемых переменных. Поэтому вызываемая функция не может изменить значения переменных в вызывающей функции, она вправе изменять только локальные временные копии этих переменных.

Таким образом, при работе с формальным параметром используется **копия фактического параметра.** Значения формальных параметров может изменяться в функции, но на значениях фактических параметров это **не отражается.**

Например, нужно составить функцию, которая меняет местами значения двух переданных в нее переменных:

```
void swap(int x, int y) { /* НЕПРАВИЛЬНО – передача по значению */
     int temp;
     temp = x;
     x = y;
     y = temp;
}
```

Если теперь вызвать данную функцию `swap` с фактическими параметрами `a` и `b`

```
int a=2, b=3;
swap(a, b);
```

то она **не измени**т значений переменных `a` и `b`.

Передача по адресу

Обойти данные ограничения позволяет передача из вызывающей функции не самих переменных, а *указателей* на эти переменные. А в самой функции соответствующие формальные параметры необходимо объявить указателями. Такой способ называют передачей ***«по адресу»***, поскольку в функцию передаются адреса переменных.

Для приведенного примера:

```
void swap(int *px, int *py) { /* ПРАВИЛЬНО – передача по адресу */
     int temp;
     temp = *px;
     *px = *py;
     *py = temp;
}
```

Теперь вызов функции `swap` с адресами переменных `a` и `b` в качестве фактических параметров

```
int a=2, b=3;
swap(&a, &b);
```

приведет к изменению значений переменных. В результате: `a=3` и `b=2`.

Примеры передачи параметров

Несколько возвращаемых значений

Способ передачи параметров по адресу используется, если необходимо, чтобы функция возвращала не одно, а несколько значений.

Допустим, требуется составить одну функцию, которая возвращает сумму и произведение двух переданных в нее чисел. Для решения этой задачи недостаточно использовать механизм `return`, который позволяет вернуть только одно значение. Организуем возврат двух значений через параметры.

Определим у функции четыре формальных параметра: два исходных числа, сумма и произведение. Первые два параметра `x` и `y` – два исходных числа – передадим по значению, поскольку внутри функции они изменяться не будут. А оставшиеся два параметра `sum` и `prod` вычисляются самой функцией, их значения изменятся, поэтому их необходимо передавать по адресу.

```
#include <stdio.h>
void calcul(int x, int y, int *sum, int *prod); /* прототип функции */
int main(void) { /* главная функция программы */
     int a, b, s, p; /* фактические параметры */
     printf("Input a: "); scanf("%d", &a);
     printf("Input b: "); scanf("%d", &b);
     calcul(a,b,&s,&p); /* вызов функции */
     printf("a+b=%d,  a*b=%d.\n", s, p);
     return 0;
}
void calcul(int x, int y, int *sum, int *prod) { /* заголовок функции */
     *sum = x + y;
     *prod = x * y;
}
```

Здесь фактические параметры `s` и `p`, переданные по ссылке, принимают новые значения после выполнения функции. При этом сама функция `calcul()` не возвращает никаких значений (`void`).

Задача об отслеживании переполнения

Рассмотрим подробно ход решения задачи отслеживания переполнения.

Пусть требуется отслеживать переполнение при арифметическом сложении двух целых положительных чисел. Составим отдельную функцию для реализации такой операции. Очевидно, что эта функция должна получать оба слагаемых, и возвращать результат сложения.

Однако операция сложения не может быть выполнена, если слагаемые настолько велики, что при их суммировании происходит переполнение типа данных, используемого для хранения результата. Поэтому кроме результата функция должна возвращать еще ***код ошибки*** (точнее, ***код завершения операции***). Например, если переполнения не произошло, то значение кода равно 0, в противном случае – оно равно 1.

Таким образом, в ходе работы такой функции *могут произойти два события:* **а)** переполнения не произошло, тогда функция возвращает код ошибки 0 и результат операции; **б)** произошло переполнение, и функция возвращает только код ошибки 1, результат операции не определен.

Теперь определим, какие параметры нужно передавать в функцию и что должна возвращать сама функция в точку вызова.

Оба слагаемых не изменяются в процессе вычисления, поэтому их можно передать в функцию в качестве двух обычных параметров, *по значению.*

Функция должна возвращать два значения – код ошибки и результат сложения. Одно из этих значений вернет сама функция, а другое придется возвращать в качестве третьего параметра, который будет передаваться в функцию *по адресу.*

Сама функция всегда возвращает некоторое значение, поэтому ***в качестве значения самой функции следует взять код ошибки,*** который тоже всегда принимает значение (0 или 1). Результат сложения не будет принимать никакого значения при переполнении, поэтому его удобнее возвращать через третий параметр, *по адресу.* Т.е. значение этого третьего параметра функция будет изменять только, если не произошло переполнения.

Реализуем описанное в программе:

```
#include <stdio.h>
#include <limits.h>   /* (содержит объявление INT_MAX) */
int add(int a, int b, int *s); /* прототип функции */
int main(void) { /* главная функция программы */
     int x, y, sum=0;
     printf("Input x: "); scanf("%d", &x);
     printf("Input y: "); scanf("%d", &y);
     if (add(x,y,&sum)) /* вызов функции, проверка кода ошибки */
          printf("Error: addition overflow.\n");
     else
          printf("Result x+y=%d\n", sum);
     return 0;
}
int add(int a, int b, int *s) { /* заголовок функции */
     if (INT_MAX-a>=b) /* INT_MAX – макс. значение типа int */
          *s = a+b;
     else
          return 1; /* возвращаемое значение кода ошибки */
     return 0; /* возвращаемое значение, если ошибки не произошло */
}
```

Обратите внимание, что в приведенном примере для использования функции *программисту нет нужды знать **как** она работает, достаточно знать **что** именно она делает.* Данным принципом следует всегда руководствоваться при реализации функций.

Поэтому, в частности, все сообщения для пользователя выводятся на экран не из вызываемой функции, а из вызывающей главной функции `main()`.

Возврат кода ошибки

Обычно значения, которые возвращает сама функция (с помощью оператора `return`), используются для отслеживания ***кодов ошибок*** выполнения этой функции.

Например, сделаем проверку переполнения типа данных `int` для обеих операций, выполняемых функцией `calcul()`.

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

```
#include <limits.h>
int calcul(int x, int y, int *sum, int *prod); /* прототип функции */
int main(void) /* главная функция программы */
{
     int a, b, s=0, p=0; /* фактические параметры */
     int error_code; /* код ошибки */
     printf("Input a: "); scanf("%d", &a);
     printf("Input b: "); scanf("%d", &b);
     error_code = calcul(a,b,&s,&p); /* вызов функции */
     if (error_code) /* проверка кода ошибки */
          printf("Overflow error: %d.\n", error_code);
     else
          printf("a+b=%d,  a*b=%d.\n", s, p);
     return 0;
}
int calcul(int x, int y, int *sum, int *prod) /* заголовок функции */
{
     if (x<=INT_MAX-y) /* проверка переполнения для сложения */
          *sum = x + y;
     else
          return 1; /* возврат кода ошибки переполнения сложения */
     if (x<INT_MAX/y+1) /* проверка переполнения для умножения */
          *prod = x * y;
     else
          return 2; /* возврат кода ошибки переполнения умножения */
     return 0; /* возврат кода успешных операций (без переполнения)*/
}
```

Теперь сама функция `calcul()` отслеживает переполнение и возвращает код ошибки той операции, при которой оно произошло (1 – при сложении, 2 – при умножении).

Использование механизма передачи параметров

Формальные и фактические параметры

Подытожим правила работы с параметрами функций.

Формальные *параметры* – это параметры, которые описываются в заголовке при определении функции, а также указываются в прототипе при описании функции.

Фактические *параметры (аргументы)* – это параметры, которые передаются в функцию при ее вызове.

В любом случае ***количество, тип и порядок следования формальных и фактических параметров должны совпадать.***

Передача по значению

Фактические параметры всегда передаются в функцию ***по значению***, т.е. значения фактических параметров копируются в ячейки памяти, выделенные при вызове функции для хранения формальных параметров. После окончания работы функции выделенная память освобождается, и значения формальных параметров теряются.

Пример передачи параметра по значению:

```
#include <stdio.h>
void func(int);  /* Описание функции (прототип). */
                 /* Имя формального параметра не указано. */
int main(void)
{
     int a = 1;  /* a – локальная переменная. */
     printf("main(): a=%d\n", a);
     func(a);  /* a передается как фактический параметр. */
     printf("main(): a=%d\n", a);
     return 0;
}
void func(int x)  /* Объявление функции (заголовок). */
                  /* x – формальный параметр. */
{  /* Тело функции */
     printf("func(): x=%d\n", x);
     x = 2;
     printf("func(): x=%d\n", x);
}
```

При вызове функции `func()` в нее передается значение переменной `a`. В формальный параметр `x` копируется значение `1`, далее функция работает с `x` как со своей локальной переменной.

Результат работы программы:

```
main(): a=1
func(): x=1
func(): x=2
main(): a=1
```

При изменении значения формального параметра `x` внутри функции `func()` не произошло изменения локальной переменной `a`, переданной в эту функцию в качестве фактического параметра. Значение формального параметра `x` потеряно после завершения работы функции `func()`.

Передача по адресу

Если требуется, чтобы функция изменяла значения передаваемых в нее фактических параметров, то эти параметры необходимо передавать ***по адресу***. При этом в функцию передаются не значения параметров, а адреса ячеек памяти, в которых они хранятся. Им соответствуют ***указатели*** в списке формальных параметров функции.

Модифицируем предыдущий пример для передачи параметра по адресу:

```
#include <stdio.h>
void func(int *);  /* Описание функции (прототип). */
                   /* Имя формального параметра не указано. */
int main(void)
{
```

```
    int a = 1;  /* a – локальная переменная. */
    printf("main(): a=%d\n", a);
    func(&a);  /* a передается как фактический параметр-адрес. */
    printf("main(): a=%d\n", a);
    return 0;
}
void func(int *x)  /* Объявление функции (заголовок). */
                   /* x – формальный параметр-указатель. */
{  /* Тело функции */
    printf("func(): x=%d\n", *x);
    *x = 2;
    printf("func(): x=%d\n", *x);
}
```

Здесь при вызове функции `func()` в нее передается не значение локальной переменной `a`, а ее адрес в памяти (`&a`, где `&` – унарная операция ***получения адреса***). Сама функция `func()` работает с формальным параметром – указателем `*x`, который принимает значение этого переданного адреса.

Поэтому для использования формального параметра `x` в теле функции `func()` необходимо помнить, что он является указателем. Для получения значения, хранящегося по этому адресу, необходимо постоянно применять к нему унарную операцию ***разыменования*** (`*x`).

Результат работы модифицированной программы:

```
main(): a=1
func(): x=1
func(): x=2
main(): a=2
```

Теперь при изменении значения формального параметра-указателя `x` внутри функции `func()` происходит соответствующее изменение локальной переменной `a`, переданной в эту функцию в качестве фактического параметра по адресу. Однако исходное значение переменной `a`, которое она принимала до вызова функции `func()`, будет потеряно.

Передача по ссылке

Передачу ***по адресу*** иногда называют передачей ***по ссылке***, хотя это не верно. На самом деле, передача по адресу является одним из способов передачи по значению, поскольку в вызываемую функцию передается значение адреса некоторой внешней области памяти с помощью операции взятия адреса (**&**). При этом в самой функции, чтобы работать с таким параметром его необходимо постоянно разыменовывать (*****).

Настоящая передача ***по ссылке*** также предполагает передачу в функцию адреса внешней переменной, но сама функция работает не с указателем, а непосредственно со значением, хранящимся по этому адресу. При этом в теле функции такую переменную не нужно разыменовывать.

Язык С не поддерживает полноценную передачу по ссылке, фактические параметры всегда передаются в функцию или ***по значению***, или ***по адресу***.

Полноценная передача параметра по ссылке реализована в таких языках программирования как, например, *Pascal* и *C++*.

Локальные и внешние переменные

Локальные переменные

Переменные, которые определены внутри тела функции являются локальными для этой функции, т.е. ни какая другая функция не может иметь к ним непосредственного доступа. Это же относится и к главной функции программы – `main()`.

Каждая локальная переменная начинает свое существование при вызове функции и прекращает его при выходе из нее. Такие переменные называются ***автоматическими***.

Для формальных параметров функций действуют те же правила, что и для локальных переменных.

Например:

```
void func(int a); /* прототип функции func() */
int main(void) {
     int x; /* x – локальная переменная для функции main() */
     x = 1;
     a = 10; /* НЕВЕРНО! Переменная a здесь недоступна */
     b = 1.23; /* НЕВЕРНО! Переменная b здесь недоступна */
     func(x); /* значение переменной x передается в func() */
     func(a); /* НЕВЕРНО! Переменная a здесь недоступна */
}
void func(int a) { /* заголовок функции func() */
     double b; /* b – локальная переменная для функции func() */
     x = 2; /* НЕВЕРНО! Переменная x здесь недоступна */
     a += 20; /* a=21, поскольку при передаче параметра a=1 */
     b = 4.56;
}
```

Если в разных функциях используются одинаковые имена локальных переменных или параметров, то они не будут связаны между собой:

```
void func(float a);
int main(void) {
     int a,b;
     a = 10; /* a – локальная переменная типа int */
     b = 20; /* b – локальная переменная типа int */
     func(1.23);
}
void func(float a) {
     double b;
     a *= 2.5; /* a – формальный параметр типа float */
     b = 4.5678; /* b – локальная переменная типа double */
}
```

При каждом вызове функции выделяется память для автоматических переменных (и формальных параметров). Выделенная память освобождается каждый раз при завершении работы функции перед возвратом в точку вызова.

При этом *переменные не сохраняют своих значений между вызовами*. Переменные должны быть явно инициализированы программистом при каждом входе в функцию. Если этого не сделать, они могут содержать случайные непредсказуемые значения.

Внешние переменные

Существует возможность определить переменные, к которым любая функция программы сможет обращаться по имени. Такие переменные будут ***внешними*** ко всем функциям.

Внешние переменные существуют постоянно – память для них выделяется при их описании, а освобождается только перед завершением работы программы.

Внешняя переменная должна быть определена только один раз за пределами всех функций программы. Пример:

```
#include <stdio.h>

int a;  /* описание внешней переменной a */
double x;  /* описание внешней переменной x */

void func1(int n);    /* прототип функции func1() */
double func2(void);   /* прототип функции func2() */

int main(void) {
     a = 2;         /* a=2, x - неопределенное */
     func1(3);      /* a=2+3=5, x - неопределенное */
     x = 1.25;      /* a=5, x=1.25 */
     x = func2(); /* a=5, x=5*1.25=6.25 */
}
void func1(int n) {
     a += n;
}
double func2(void) {
     return a*x;
}
```

Если описывать внешние переменные в самом начале программы, то они будут доступны во всей программе глобально, и к ним сможет иметь доступ ***любая функция в любой момент***.

Внешние переменные часто используются вместо аргументов для обмена данными между функциями.

Побочные эффекты

Порядок вычисления операндов в операциях

В языке С для большинства операций ***не определен порядок вычисления операндов***. Например, в следующем выражении

```
s = func1() + func2();
```

первой может вызываться как функция `func1()`, так и функция `func2()`.

Например, если от внешней переменной `a`, которую изменяет одна из функций `func1()`, зависит работа другой функции `func2()`, то результат будет зависеть от порядка вызова функций:

```
#include <stdio.h>

int a;  /* описание внешней переменной a */

int func1(void);  /* прототип функции func1() */
int func2(void);  /* прототип функции func2() */

int main(void) {
     int s;
     a = 1;  /* изменяется внешняя переменная a=1 */
     s = func1() + func2();  /* НЕКОРРЕКТНОЕ ВЫРАЖЕНИЕ! */
     printf("Sum = %d\n", s);
}
int func1(void) {
     a = 2;  /* изменяется внешняя переменная a=2 */
     return 0;
}
int func2(void) {
     if (a%2) /* результат зависит от внешней переменной a */
          return 1;
     else
          return 0;
}
```

Поскольку порядок вызова функций не определен, результат (`s=0` или `s=1`) будет зависеть от компиляции в конкретной системе. Однако программа будет всегда успешно компилироваться и выполняться.

Подобные явления называются ***побочными эффектами***. ***При составлении программы их необходимо избегать.***

Чтобы гарантировать нужный порядок вычислений программисту следует ***явно определить последовательность вызова функций.*** Например, введя дополнительные переменные:

```
int main(void) {
     int s, s1, s2;  /* дополнительные переменные s1 и s2 */
     a = 1;
     s1 = func1();  /* первой вызывается функция func1() */
     s2 = func2();  /* второй вызывается функция func2() */
     s = s1 + s2;  /* выражение не зависит от порядка вызова */
     printf("Sum = %d\n", s);
}
```

Порядок вычисления аргументов функции

Аналогично, в языке С ***не определен порядок, в котором вычисляются аргументы функции при ее вызове.***

Например, следующий код может дать разные результаты при компиляции в различных системах:

```
int n = 8;
printf("%d %f \n", ++n, sqrt(n)); /* НЕКОРРЕКТНЫЕ АРГУМЕНТЫ! */
```

Здесь результат зависит от того, когда переменная `n` получает приращение, до или после вызова функции `sqrt()`.

Этот пример также демонстрирует ***побочный эффект***, *которого необходимо избегать.* Программисту следует ***самому определить нужный порядок вычислений***. Например, так:

```
int n = 8;
++n;  /* сначала приращение n */
printf("%d %f \n", n, sqrt(n)); /* затем вызов sqrt(n) */
```

Пример использования функций

Задача о числах в разных системах счисления

Для выполнения заданий для лабораторной и самостоятельной работы можно воспользоваться приведенной ниже функцией, которая переводит числа из системы счисления с основанием `BASE` в десятичную систему. В данном случае исходная система счисления – троичная (`BASE=3`):

```
#include <stdio.h>
#define BASE 3

long int BASEto10(long int a);

int main(void) {
     long int x, y;
     printf("Ternary notation: (input only digits 0,1,2) a=");
     scanf("%ld", &x);
     y = BASEto10(x);
     printf("Decimal notation: a=%ld\n", y);
     return 0;
}
long int BASEto10(long int a) {
     int k=1;
     long int a10=0;
     while (a) {
          a10 += k*(a%10);
          k *= BASE;
          a /= 10;
     }
     return a10;
}
```

Рекурсия

Понятие рекурсии

Рекурсивным называется объект, который частично определяется через самого себя. В программировании на языке С ***рекурсия*** – способ организации вычислительного процесса, при котором функция **вызывает сама себя.**

Рекурсивные определения широко используются во многих областях, особенно в математике.

Рассмотрим функцию факториала ***n*!**. Как правило, ее определяют как произведение первых ***n*** целых чисел:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$$

Такое произведение можно вычислить с использованием итеративной конструкции цикла (это решение приводилось ранее, см. ***тематическое занятие 2***):

```
#include <stdio.h>
int main(void) {
     int n, i;
     long int fact=1;
     printf("Input n:");
     scanf("%d", &n);
     for (i=1; i<=n; i++)
          fact*=i;
     printf("Factorial n!=%ld\n", fact);
     return 0;
}
```

Однако существует также другое (рекурсивное) определение факториала, в котором используется ***рекуррентная*** формула:

(1) **0! = 1**
(2) для любого $n > 0$ $n! = n \cdot (n - 1)!$

Этому определению факториала соответствует программа, использующая рекурсивную функцию, которая вызывает сама себя:

```
#include <stdio.h>

long int fact(int i);

int main(void) {
     int n;
     printf("Input n:");
     scanf("%d", &n);
     printf("Factorial n!=%ld\n", fact(n));
     return 0;
}
long int fact(int i) {
     if (i==1)  /* условие останова */
          return 1;
     else
          return i*fact(i-1);  /* рекурсивный вызов */
}
```

Следует обратить внимание, что здесь совсем **нет операторов цикла.** При этом операция умножения будет **повторяться *n*–1 раз,** поскольку функция `fact` вызывает саму себя *n*–1 раз.

Заметьте, что использование рекурсии позволяет легко (почти дословно) запрограммировать вычисления по рекуррентным формулам.

Разбор рекурсии на примере

Разберем подробнее приведенный пример рекурсивного вычисления факториала ***n*!**.

При каждом ***рекурсивном вызове*** `fact(i-1)` выполнение текущей функции приостанавливается, но ее переменные не удаляются из памяти. Происходит новый вызов функции, для переменных которой также выделятся память, и так далее. Образуется последовательность прерванных процессов, из которых выполняется всегда последний. Передаваемый в функцию параметр каждый раз уменьшается на `1`.

Когда параметр станет равным `1`, выполнится ***условие останова*** `i==1`. Рекурсивные вызовы закончатся, и функция вернет значение равное `1` (`return 1;`). Текущий вызов функции окажется последним. Затем в обратном порядке последовательно закончат работу все остальные вызванные функции.

Последняя завершенная функция (она же – первая вызванная) вернет результат, который будет выведен на экран стандартной функцией `printf`.

Глубина и уровень рекурсии

Максимальное число рекурсивных вызовов функции без возвратов называется ***глубиной*** рекурсии. Число рекурсивных вызовов в каждый конкретный момент времени называется ***текущим уровнем*** рекурсии.

В рассмотренном примере, если пользователь ввел число `n`=5, то общее количество вызовов функции `fact` равно 5, а глубина рекурсии (количество рекурсивных вызовов) равна 4. При первом вызове функции `fact` (в качестве одного из параметров функции `printf`) текущий уровень рекурсии считается равным нулю, т.к. функция еще не успела вызвать саму себя.

Поскольку имена параметров и локальных переменных не меняются при каждом рекурсивном вызове, то ***воспользоваться значением некоторой переменной i-го уровня рекурсии можно, находясь только на этом i-м уровне.*** В рассмотренном примере у функции `fact` нет локальных переменных, но есть параметр `i`, область видимости которого соответствует данному правилу.

Таблица трассировки

Для демонстрации действий, выполняемых рекурсивной функцией, используют ***таблицу трассировки*** значений ее параметров по уровням рекурсии.

В рассмотренном примере рекурсивного вычисления факториала действия выполняются на рекурсивном возврате, и при вводе пользователем `n`=5 таблица трассировки:

Текущий уровень рекурсии	Рекурсивный спуск	Рекурсивный возврат
0	Ввод: `n=5` `fact(5)`	Вывод: `n!=120`
1	`i=5` `fact(4)`	`return 5*24; (fact=120)`
2	`i=4` `fact(3)`	`return 4*6; (fact=24)`
3	`i=3` `fact(2)`	`return 3*2; (fact=6)`
4	`i=2` `fact(1)`	`return 2*1; (fact=2)`
5	`i=1` `return 1; (fact=1)`	

Недостатки и достоинства рекурсии

Для каждого нового рекурсивного вызова функции выделяется память, которая освобождается только после достижения всей глубины рекурсии. Поэтому выполнение рекурсивных функций требует **значительно большего объема оперативной памяти** во время выполнения программы, чем нерекурсивных. Если глубина рекурсии очень велика, то это может привести к **переполнению памяти.**

К тому же рекурсивные алгоритмы не дают выигрыша в быстродействии, а, как правило, выполняются **медленнее.**

Однако код программы с рекурсией обычно компактнее, и его значительно легче писать и дорабатывать. Рекурсия удобна при работе с рекурсивно определенными алгоритмами и структурами данных.

Формы рекурсивных функций

Условие останова

Рекурсивное определение позволяет с помощью конечного выражения определить бесконечное множество объектов. А с помощью рекурсивного алгоритма можно определить **бесконечное вычисление,** причем алгоритм не будет содержать повторений фрагментов кода.

Рекурсивные функции, не содержащие ***условия останова*** – ***прекращения рекурсивных вызовов,*** – приводят к бесконечным процессам.

Например:

```
#include <stdio.h>

void endless(void);

int main(void) {
     endless();
     return 0;
}
void endless(void) {
     printf("Функция вызвана. Снова вызываем функцию:\n");
     endless();
}
```

Действие программы прервется, когда будет исчерпана вся свободная память. Использовать подобные рекурсивные функции **невозможно** из-за ограниченности оперативной памяти.

Поэтому **главное требование** к рекурсивным функциям заключается в том, что ***вызов рекурсивной функции должен выполняться по условию, которое на каком-то уровне рекурсии станет ложным.***

Если это условие истинно, то ***рекурсивный спуск*** продолжается. А если условие становится ложным, то спуск заканчивается и начинается поочередный ***рекурсивный возврат*** из всех вызванных на данный момент копий рекурсивной функции.

Структуры рекурсивных функций

Структура рекурсивной функции может принимать **три** различные формы. Рассмотрим их на примере функции `rec()` без параметров, которая не возвращает никакого значения.

1) *Выполнение действий на рекурсивном спуске*

```
void rec(void) {
     ...
     Операторы;/* вначале выполняются действия */
     ...
     if (Условие) rec(); /* затем происходит рекурсивный вызов */
}
```

2) *Выполнение действий на рекурсивном возврате*

```
void rec(void) {
     if (Условие) rec(); /* вначале происходит рекурсивный вызов */
     ...
     Операторы;/* затем выполняются действия */
     ...
}
```

3) *Выполнение действий как на рекурсивном спуске, так и на рекурсивном возврате*

```
void rec(void) {
     ...
     Операторы1;/* вначале выполняется 1-я часть действий */
     ...
     if (Условие) rec(); /* потом происходит рекурсивный вызов */
     ...
     Операторы2;/* затем выполняется 2-я часть действий */
     ...
}
```

или то же самое, но с проверкой условия вначале:

```
void rec(void) {
     if (Условие) { /* проверка условия останова */
          ...
          Операторы1; /* 1-я часть действий */
          ...
          rec(); /* рекурсивный вызов */
          ...
          Операторы2; /* 2-я часть действий */
          ...
     }
}
```

Многие задачи безразличны к тому, какая форма рекурсивных функций используется. Но существуют классы задач, при решении которых требуется сознательно управлять ходом работы рекурсивных функций.

Упражнения

Упражнение 5.1

Составить программу, которая содержит функцию, переводящую размер в дюймах в метры. Пользователь вводит размер в дюймах, вызывается функция, которая переводит его в метры. Результат, возвращаемый функцией, выводится на экран в основной программе.

Упражнение 5.2

Составить программу, которая содержит функцию, вычисляющую периметр и площадь прямоугольника. Пользователь вводит размеры двух сторон прямоугольника ***a*** и ***b***. Вызывается функция, которая вычисляет периметр ***P*** и площадь ***S*** прямоугольника. Значения ***P*** и ***S***, возвращаемые функцией, выводится на экран в основной программе.

Упражнение 5.3

Реализовать проверку переполнения при вычислении значений ***P*** и ***S*** в функции, описанной в упражнении ***5.2***.

Упражнение 5.4

Приведенную программу рекурсивного вычисления факториала дополнить выводом таблицы трассировки на экран. Обязательные поля таблицы: *текущий уровень рекурсии, значения на рекурсивном спуске, значения на рекурсивном возврате.*

Упражнение 5.5

Составить программу, которая, используя рекурсивную функцию, находит значение данной функции для любых целых неотрицательных аргументов ***n*** и ***a***:

$$R(n,a)=\underbrace{\sqrt{a+\sqrt{a+\ldots+\sqrt{a}}}}_{n\text{-корней}}$$

Вывести на экран таблицу трассировки.