Функции

Вызов функции - это оператор. У вызова функции есть приоритет – один из самых высоких. Список аргументов функции считают за один операнд, так что оператор оказывается бинарным (первый операнд - сама функция, второй - список ее аргументов).

Пример записи функции func:

```
double func(double param1, int param2)
{  return param1-0.1*param2; }
```

Сначала указан тип значения, которое функция возвращает - в данном случае это *double*. Затем после пробела следует имя функции - идентификатор, составленный по тем же правилам, что и для имен переменных. После имени функции в круглых скобках перечислены формальные параметры с указанием их типов.

Функции

Формальные параметры разделены запятыми. В нашей функции это *param1* типа *double* и *param2* типа *int*.

После круглых скобок со списком формальных параметров следует блок с телом функции - тот, который в фигурных скобках, причем в теле функции мы можем использовать формальные параметры как обычные переменные.

Функции

Определив функцию, мы можем ее неоднократно вызывать, задавая в
качестве фактических параметров нужные нам переменные или
значения. При этом мы можем использовать то значение, которое она
возвращает, а можем его игнорировать (если нам просто надо, чтобы
ыполнились операторы в теле функции).

```
int i;
    double x, result;

    ...

    /* два вызова функции в выражении */
    result = func(x,i) * func( i+x, 100 );

    /* Вызываем еще раз, но игнорируем возвращаемое значение */
    func(x, i);
```

Функции без возвращаемого значения

```
/* У этой функции нет возвращаемого значения */
    void f() {
      ...
      return;
    }
```

В операторе *return* нет никакого значения, сразу после ключевого слова стоит точка с запятой.

Также можно написать *void* вместо списка параметров, если функции параметры не нужны:

```
int f(void) {
      ...
      return 0;
    }
```

Параметры и переменные

```
int i, j;

    /*   У первой функции видны i, j файлового уровня.  Кроме того, у
    нее есть формальный параметр k  и локальная переменная result   В
    процессе работы эта функция изменяет значение файловой
    переменной i */

    int f1(int k) {
      int result;
      result = i*j + k;
      i += 100;
      return result;
    }
```

Параметры и переменные

```
/*  Во второй функции имя формального параметра совпадает с
    именем переменной i файлового уровня, при
    работе   используется параметр, а не файловая переменная. */

    int f2(int i)
{
     /* i - параметр, j - файловая */
     return i*j;
}
```

Параметры и переменные

```
/*  С третьей функцией аналогичная ситуация, что и со
    второй.  Только на этот раз маскируется файловая  переменная j,  и
    не формальным параметром, а локальной переменной. */

    int f3(int k)
{
     int j;
     j=100;
     /* i - файловая, j - локальная */
     return i*j + k;
}
```

Параметры и переменные

```
Переменная j самого внутреннего блока  маскирует не только
    файловую, но и локальную переменную из  внешнего блока. */
    int f4 (int k)
{
     /* Объявляем переменную и сразу инициализируем */
     int j=100;
     {
       /*   Объявляем еще одну локальную с тем же именем,  что у
    файловой и локальной из внешнего блока */
       int j=10;
       /*   i - файловая, j - локальная, причем из    внутреннего
    блока  */
       return i*j + k;
     }
}
```

Необходимость инициализации переменных (автоматические переменные)

```
/* Файловая переменная без инициализации, будет равна 0 */
    int s;

    int f() {
      /* Локальная без инициализации, содержит "мусор" */
      int k;
      return k;
    }

    int main() {

      printf("%d\n", s); /* Всегда печатает 0 */

      /* Невозможно предсказать, что увидим */
      /* К тому же числа могут быть разными */
      printf("%d\n", f());
      ...;
      printf("%d\n", f());
      return 0;
```

Статические переменные

```
int f() {
    static int i;
    return i;
  }
```

Перед обычным определением переменной модификатор типа – ключевое слово *static*. Теперь функция всегда возвращала бы 0 – локальные статические также, как и файловые, создаются один раз и инициализируются нулем, если только не задать другую инициализацию.

Эти переменные создаются один раз за время работы программы, и один раз инициализируются - либо нулем, либо тем значением, которое было задано. Поскольку они «живут» независимо от функции, значит в одном вызове функции в такую переменную можно что-то положить, а в следующем - это что-то использовать.

Статические переменные

```
int f() {
        static int ncalls=1;

        /* Который раз мы эту функцию вызвали? */
        printf("number of calls %d\n", ncalls++);

        ...
      }
```

Статические переменные

Полезный «трюк», основанный на статических локальных переменных –
возможность выполнять какие-то дорогостоящие «подготовительные» операции
только один раз.

```
int func() {
  /*
    Неявная инициализация тоже дала бы 0, но правила хорошего тона
требуют ...  */
  static int init_done=0;

  if (!init_done) {
    /* Здесь мы выполняем какую-то "дорогостоящую«,  но разовую
работу - например, считываем  таблицу значений из файла. А потом
указываем,  что таблица прочитана и при следующих вызовах   этого
делать уже не нужно.  */
    read_table();
    init_done = 1;
  }

  /*
```

Передача по значению

Передача параметра по значению" и "передача параметра по
ссылке".

```
#include <stdio.h>

    void f(int k) {
    k = -k;
    }

    int main() {
      int i = 1;
      f(i);
      printf("%d\n", i);
      return 0;
    }
```

Адреса и указатели

```
int i;
    double d;
    /*
     Функции передаются адреса переменных i и d.  После вызова функции
адреса останутся прежними (pass-by-value), но значения могут измениться */
    func( &i, &d );
    ...
char *s; /* указатель на char */

    int *pi; /* указатель на int */
    void *pv; /* указатель на void */
    char **av /* указатель на указатель на char */

    /*   Это указатель на функцию, которая возвращает   int, а в качестве
    параметра ожидает char */
    int (*pf)(char)
```

Передача параметров по значению

```
void func(int *p1, double *p2){

    /*  Засылаем в целую переменную, на которую  указывает p1, значение
   1 */
    *p1 = 1;

    /*  Добавляем 10.5 к переменной типа double,    на которую указывает
   p2. */
    *p2 += 10.5;
   }
```

При вызове подобной функции не обязательно указывать в качестве параметра

адрес какой-то переменной, можно вместо этого поставить все тот же указатель,

в котором такой адрес содержится. Так, например, следующие два вызова нашей

функции func приведут к одному и тому же результату:

- ```
  int i, *pi;
  double d, *pd;

  /* Указываем непосредственно адреса */
  ```
  ```

Указатели можно использовать не только для параметров, но и в качестве возвращаемого значения функции. Вот как, например, выглядит
определение и вызов функции, возвращающей указатель на тип char (такой тип используется для передачи текстовых строк).

```
char *genstr() {
      char *p;
      ...
      return p;
    }

char *s;

s = genstr();
```

Чем «опасны» указатели?

```
void f(int *p) {
    *p=1;
}

int main() {
  int i;
  int *ptr;

  f(ptr);

  ...
  return 0;
}
```

Что делать, если указатель создан, но пока не известно, какой адрес в него записать? Для этого есть специальное значение указателя - в C это символьная константа *NULL*, в C++ - 0.

```
#include <stdlib.h>
    main() {
      char *p=NULL;
      ...}

void f(int *p) {
      if (p != NULL) *p=1;
      else {  printf("Help!!! NULL pointer in f()\n"); abort(); }
    }

int main() {
      int i;  int *ptr=NULL;  f(ptr);
      ...
      return 0;
}
```

Ввод-вывод

```
#include <stdio.h>
    int main() {
      /* Печатаем целое число и строку */
      printf("integer=%d, string=%s \n", 10, "hello");
    }
```

```
#include <stdio.h>
    int main() {
      int i;
      char c;
      /* Вводим целое число и символ */
      scanf("%d %c", &i, &c);
    }
```

scanf() - чтение данных из потока

```
int i, j;
    scanf("%d %d", &i, &j);

// или
int i, j;
    int *p1,*p2;
    p1=&i;
    p2=&j;
    scanf("%d %d", p1, p2);
```

Объявление и определение функции

```
int main() {
        int i;

        i = 1;

        f(i);
        ...
        return 0;

void f(double x) {
        ...
      }

void f(double x);
      int main() {
        ...
```

Ввод-вывод - <stdio.h>

FILE *fopen(char *filename, char *mode)
открывает поток, связанный с файлом *filename*. Режим работы определяется строкой *mode*. Например "r" - открыть на чтение, "w" - на запись. При успешном завершении возвращает указатель на открытый поток, при ошибке - *NULL*.

int fflush(FILE *f)
записывает все накопленные в буфере выходного потока *f* данные в файл. *fflush(NULL)* выполняет эту операцию со всеми открытыми выходными потоками. При успешном завершении возвращает 0, при ошибке - *EOF*.

int fclose(FILE *f)
закрывает поток f. При успешном завершении возвращает 0, при ошибке - *EOF*.

Ввод-вывод - <stdio.h>

- FILE *tmpfile()
  создает временный файл для записи. Файл будет автоматически удален по *fclose()* либо при завершении программы. При успешном завершении возвращает указатель на открытый поток, при ошибке - *NULL*.

  char *tmpnam(NULL);
  char *tmpnam(char result[]);

  возвращает уникальное имя временного файла (сам файл не создается). Без аргумента возвращает указатель на статическую строку. С аргументом - копирует имя в указанный массив, и возвращает указатель на этот массив.

  int printf(char *fmt, ...)
  int fprintf(FILE *f, char *fmt, ...)
  int sprintf(char *buf, char *fmt, ...)

  Функции форматного вывода соответственно в *stdout*, в поток *f* и в символьный массив *buf*. При успешном завершении возвращают число выведенных символов (возможно, 0). При ошибке - отрицательное значение.

Ввод-вывод - <stdio.h>

- int scanf(char *fmt, ...)
  int fscanf(FILE *f, char *fmt, ...)
  int sscanf(char *buf, char *fmt, ...)

  Функции форматного ввода соответственно из *stdin*, из потока *f* и из символьного массива *buf*. При успешном завершении возвращают число успешно введенных элементов (не символов, а элементов, введенных по спецификациям, заданным в форматной строке). При ошибке возвращают *EOF*.

  int getchar()
  int fgetc(FILE *f)

  Считывают одиночный символ из *stdin* (*getchar()*) или из входного потока *f* (*fgetc()*). Возвращают либо символ в виде *unsigned char* (неотрицательный результат), либо *EOF*, если поток исчерпан и при ошибке.

  int putchar(int c)
  int fputc(int c, FILE *f)

  Записывают одиночный символ в *stdout* (*putchar()*) или в выходной поток *f* (*fputc()*). Возвращают переданный в поток символ либо, при ошибке, *EOF*.

Ввод-вывод - <stdio.h>

- int ungetc(int c, FILE *f)

  "Посмотрели на символ - не понравился". Функция отправляет символ c обратно во **входной** поток *f*. Стандарт гарантирует возвращение только одного символа (некоторые реализации позволяют и больше). Функция возвращает переданный обратно в поток символ либо, при ошибке, *EOF*.

  char *gets(char *buf)

  Считывает строку (от начала строки до символа '\n') из *stdin* в символьный массив *buf*. Символ \n из строки удаляется, точнее, заменяется нулевым байтом. Функция потенциально опасна - не позволяет защититься от ввода строк, длина которых превышает размер массива *buf*. При успешном вводе возвращает *buf*. По исчерпанию ввода и при ошибке - *NULL*.

  char *fgets(char *buf, int size, FILE *f)

  Ввод строки из потока *f*. Работает аналогично *gets*. Отличия: - во-первых, вводит из потока не более *size-1* символов (защита от переполнения строки *buf*); во вторых - не удаляет из строки символ '\n' (но нулевой байт в конец строки добавляет)

Ввод-вывод - <stdio.h>

- int puts(char *s)
  int fputs(char *s, FILE *f)

  Выводят С-строку в *stdout* (*puts()*), либо в поток *f* (*fputs()*). Возвращают неотрицательное значение при успешном завершении , либо *EOF* при ошибке.

Ввод-вывод - <stdio.h>

- int fseek(FILE *f, long offset, int fromwhere)

  Позиционирование в файле (функция смещает указатель записи-чтения для данного файла). Второй аргумент указывает, на сколько надо сместиться. Третий аргумент позволяет выполнять смещение от начала или конца файла, либо от текущей позиции. Возвращает 0 при успешном завершении, -1 при ошибке.

  long ftell(FILE *f)

  Позволяет получить текущую позицию указателя записи-чтения в открытом файле *f*. При ошибке возвращает -1.

  void rewind(FILE *f)

  Устанавливает указатель записи-чтения файла на начало. Ничего не возвращает.

  size_t fread(void *ptr, size_t size, size_t nobj, FILE *f)
  size_t fwrite(void *ptr, size_t size, size_t nobj, FILE *f)

Ввод-вывод - <stdio.h>

- size_t fread(void *ptr, size_t size, size_t nobj, FILE *f)
  size_t fwrite(void *ptr, size_t size, size_t nobj, FILE *f)

  Функции позволяют читать из потока, либо записывать в него произвольные данные. При этом первый аргумент - указатель на массив, куда данные надо передавать из потока, либо откуда брать для записи в поток. Второй аргумент - *size* - задает размер одного элемента, третий - максимальное число объектов, которое можно считать, либо записать.
- Функция *fread()* возвращает количество считанных объектов. Чтобы понять, была ли при вводе ошибка и был ли исчерпан поток *f*, надо использовать функции *ferror()* и *feof()*.
- Функция *fwrite()* также возвращает количество объектов (только не считанных, а записанных). Но об ошибке можно догадаться без *ferror()*, поскольку функция вернет меньшее значение, чем указано в *nobj*.

Ввод-вывод - <stdio.h>

- int feof(FILE *f)

  возвращает ненулевое значение, если у потока установлен индикатор конца файла (например, если при последнем вводе из потока данные были исчерпаны).

  int ferror(FILE *f)

  возвращает ненулевое значение, если у потока установлен индикатор ошибки (при последней операции с потоком возникла ошибка).

  void clearerr(FILE *f)

  Очищает в потоке *f* индикаторы конца файла и ошибки.

  void perror(char *header)

  Выводит внятное сообщение в поток *stderr*.

Работа со строками - <string.h>

Функции с именами, начинающимися с *str*, работают с С-строками, в которых нулевой байт означает конец строки. Функции же, начинающиеся с *mem*, работают с массивами символов, следовательно, позволяют работать и с нулевыми байтами.

Функции копирования и слияния строк (модифицирующие один из аргументов) изменяют первый аргумент, а не второй.

char *strcpy(char *s1, char *s2)
копирует строку s2 в s1. Возвращает s1.

char *strncpy(char *s1, char *s2, size_t n)
копирует строку s2 в s1, но копируется не более n символов. Возвращает s1.

char *strcat(char *s1, char *s2)
объединяет строки s1 и s2 (дописывает s2 в s1). Возвращает s1.

char *strncat(char *s1, char *s2, size_t n)
объединяет строки s1 и s2, но дописывает не более n символов. Возвращает s1.

Работа со строками - <string.h>

int strcmp(char *s1, char *s2)

сравнивает строки (содержимое, не указатели), расставляя их в лексикографическом порядке. Возвращает 0 для совпадающих строк, отрицательное значение при s1<s2 и положительное при s1>s2.

int strncmp(char *s1, char *s2, size_t n)

тоже сравнивает строки, но берет из них для сравнения не более n первых символов.

char *strchr(char *s, int c)

возвращает указатель на первый встреченный в строке символ c. Если такого символа в строке не оказалось, возвращает NULL.

char *strrchr(char *s, int c)

похожа на strchr(), но возвращает указатель на последний символ c в строке.

char *strstr(char *s1, char *s2)

возвращает указатель на первую встреченную в строке s1 подстроку s2. Если подстроки не нашлось, возвращает NULL.

size_t strlen(char *str)

возвращает длину строки.

Работа со строками - <string.h>

char *strerror(size_t n)
возвращает строку сообщения, соответствующего ошибке с номером n.

void *memcpy(void *dst, void *src, size_t len)
копирует len байтов (включая нулевые) из src в dst. Возвращает dst.

void *memove(void *dst, void *src, size_t len)
делает то же, что и memcpy. Это - единственная функция, которая по стандарту обязана правильно копировать перекрывающиеся объекты.

int memcmp(void *s1, void *s2, size_t len)
аналог strcmp, но с учетом нулевых байтов.

void *memchr(void *s, int c, size_t len)
аналог strchr, но с учетом нулевых байтов.

void *memset(void *s, int c, size_t len)
заполняет первые len байтов массива s символом c.

Математические функции - <math.h>

double sin(double x) - синус

double cos(double x) - косинус

double tan(double x) - тангенс

double asin(double x) - арксинус

double acos(double x) - арккосинус

double atan(double x) - арктангенс

double atan2(double y, double x) - Арктангенс y/x. В отличие от обычного, определяет по знакам y и x квадрант и возвращает значение в диапазоне от -pi до pi (обычный - от -pi/2 до pi/2)

double sinh(double x) - гиперболический синус double cosh(double x) - гиперболический косинус

double tanh(double x) - гиперболический тангенс

Математические функции - <math.h>

double exp(double x) - e в степени x

double log(double x) - натуральный логарифм

double log10(double x) - десятичный логарифм

double pow(double x,double y) - x в степени y

double sqrt(double x) - квадратный корень

double fabs(x) - абсолютное значение x (модуль x)

double ceil(double x) - наименьшее целое, которое не меньше x, приведенное к типу double

double floor(double x) - наибольшее целое, которое не больше x, приведенное к типу double

double hypot(double x, double y) - длина гипотенузы.

double modf(double x, double *ip) - разбивает число на целую и дробную части. Дробная часть возвращается в качестве результата, целая записывается в ip.

Функции общего назначения - <stdlib.h>

Объявления функции для работы с динамической памятью в С-стиле

```
void *malloc(size_t size);
    void *calloc((size_t number, size_t size);
    void *realloc(void *ptr, size_t size);
    void *free(void *ptr);
```

Функции общего назначения - <stdlib.h>

- double atof(char *s)
  int atoi(char *s)
  int atol(char *s)
  double strtod(char *s, char **endp)
  long strtol(char *s, char **endp, int base)
  unsigned long strtoul(char *s, char **endp, int base)

  Эти функции позволяют преобразовывать символьному выражение числа, хранящееся в строке, в число. Это одна из альтернатив *scanf()* - возможно сначала считать из входного потока строку, а потом уже из этой строки с помощью таких функций извлекать значения переменных.
- Наиболее широкие возможности у последних трех - *strto...()*. Во первых, они позволяют работать с системами счисления от двоичной до 36-ричной (разумеется, это не касается функции *strtod()*), система задается аргументом *base*; а во вторых, прочитав из строки число, эти функции возвращают в *endp* адрес следующего символа, чтобы обеспечить возможность продолжить с этого места разбор строки.

Функции общего назначения - <stdlib.h>

- int rand()
  void srand(unsigned int seed)
  long random(unsigned long seed)
  void srandom(unsigned long seed)

  Работа с псевдослучайными числами. Функции *rand()* и *random()* генерируют случайные числа, равномерно распределенные по всему диапазону типа *int* и *long*, соответственно. Функции *srand()* и *srandom()* позволяют, задавая разные значения *seed*, получать различные последовательности чисел (очередное псевдослучайное числа на самом деле рассчитываются по рекурсивным алгоритмам, так что, стартуя генератор с одного и того же места, можно будет получать одинаковую последовательность). Пользоваться лучше парой функцией random() - у нее по сравнению с *rand()* шире диапазон, и гораздо больше период последовательности.

Функции общего назначения - <stdlib.h>

- int abs(int n)
  long labs(long n)

  Эти две функции возвращают абсолютное значение числа типа int и long соответственно.

  div_t div(int num, int denom)
  ldiv_t ldiv(long num, long denom)

  Функции вычисляют частное и остаток от деления для чисел int и long, возвращая оба результата в полях структуры типа div_t или ldiv_t соответственно.

  void abort()

  Аварийное завершение задачи с созданием образа памяти для последующей отладки.

Функции общего назначения - <stdlib.h>

- void exit(int status)

  Нормальное завершение задачи. Значение аргумента *status* передается операционной системе. В unix-подобных операционных системах успешным завершением принято считать возврат нулевого статуса. Возврат значения из *main()* с помощью оператора return полностью эквивалентен вызову *exit()* с этим значением.

Функции общего назначения - <stdlib.h>

- int atexit(void (*func)())

  Позволяет назначать функции, которые будут вызваны при нормальном завершении задачи.

  int system(const char *s)

  По этому вызову программа запустит команду операционной системы, заданную в строке s, дождется ее завершения, а затем продолжит работу.

  char *getenv(char *name);
  int setenv(const char *name, const char *value, int overwrite);
  int putenv(const char *string);
  void unsetenv(const char *name);

  Эта группа функций позволяет работать с переменными окружения (например, с переменной PATH) - получать и менять их значения, создавать и уничтожать переменные. Переменные окружения - это те строки, в которых сама операционная система хранит различные параметры, нужные для нормального запуска и функционирования программ. Не следует путать их с переменными самой программы.

Функции общего назначения - <stdlib.h>

- void *bsearch(void *key, void *base, size_t nelm, size_t size, int (*compare)(void *keyval, void *data));

Эта функция осуществляет поиск заданного элемента key в массиве любого типа base. nelm и size задают число элементов и размер элемента. Если найдет - возвращает указатель на него. Функцию, сравнивающую два элемента, необходимо написать самостоятельно - именно поэтому у последнего аргумента bsearch такой тип.

void (void *base, size_t nelm, size_t size, int (*compare)(void *keyval, void *data));

Эта функция сортирует массив элементов произвольного типа. Смысл аргументов такой же, как и у функции bsearch().

Работа с датой и временем - <time.h>

- clock_t clock()
  Возвращает время процессора, использованное программой с момента запуска. Время измеряется в долях секунды, равных 1/CLOCKS_PER_SEC.

  t time(time_t *t)
  Возвращает время в секундах, прошедшее с начала эпохи UNIX (с 0 часов 0 минут по Гринвичу 1 января 1970 года). Если t - ненулевой указатель, то время записывается и в t тоже.

  difftime(time_t t1, time_t t2)
  Вычисляет разницу t1-t2.

  ctime(time_t *t)
  Возвращает строку, в которой записана дата, соответствующая заданному аргументом времени. Учитывает часовой пояс. Иными словами, написав в программе
  time_t t;
  time(&t);
  printf("%s\n", ctime(&t);
  в *stdout* помещается строка с местным временем.