Функции

Подпрограммы

Подпрограмма - именованная часть программы, содержащая описание определённого набора действий. Подпрограмма может быть многократно вызвана из разных частей программы.

Функция - это подпрограмма специального вида, которая всегда должна возвращать результат. Вызов функции является, с точки зрения языка программирования, выражением, он может использоваться в других выражениях или в качестве правой части присваивания.

Процедура - это независимая именованная часть программы, которую после однократного описания можно многократно вызвать по имени из других частей программы для выполнения определенных действий.

Преимущества подпрограмм

- Уменьшение сложности программирования.
- Модульность (использование подпрограмм позволяет разделить большую программу на части меньшего размера, которые легче понимать и изменять).
- Закрытость реализации (изменение реализации подпрограммы не влияет на остальную часть программы, если интерфейс подпрограммы не изменился).
- Расширение возможностей языков программирования (создание библиотек).

Подпрограммы в Си

В языке Си подпрограммы представлены только функциями.

```
// заголовок функции
тип-результата имя-функции(список формальных параметров
                                                с их типами)
// тело функции
{
    определения
    выражения
}
```

```
float avg(float a, float b)
{
    float c;
    c = a + b;
    return c / 2.0;
}
```

Возвращаемое значение

- Функция может вернуть значение любого типа кроме массива.
- Если функция ничего не возвращает, то в качестве типа возвращаемого значения следует указать void.
- Если тип возвращаемого значения не указан, то согласно стандарту C89, компилятор предполагает, что возвращается значение целого типа.
- Согласно стандарту C99 тип возвращаемого значения опускать нельзя (warning).
- Для возврата значения используется оператор return.

Параметры функции

- Любая функция может принимать параметры.
- Если список параметров содержит только ключевое слово void, у функции нет параметров.
- Формальные параметры перечисляются через запятую.
- Определение параметра начинается с указания его типа, за которым следует имя параметра.
- Для каждого имени тип указывается отдельно.

Тело функции

- У каждой функции есть исполнимая часть, которая называется *телом функции* и заключена в фигурные скобки, которые также являются частью тела функции.
- Тело функции может содержать как объявление переменных, так и операторы.
- Переменные, описанные в теле функции, «принадлежат» только этой функции и не могут быть ни прочитаны, ни изменены другой функцией. Тело функции не может содержать в себе определения других функций.
- Если функция ничего не возвращает, ее тело может быть пустым.

Оператор return

return *выражение*;

- Оператор return завершает выполнение функции и возвращает управление вызывающей стороне.
- Функция может содержать произвольное число операторов return.
- Оператор return может использоваться в функциях типа void. При этом никакое выражение не указывается.

Оператор return

```
int max(int a, int b)
{
    if (a < b)
        return b;

    return a;
}
```

```
void print_pos(int a)
{
    if (a < 0)
        return;

    printf("%d\n", a);
}
```

```
void beep(void)
{
    printf("\a");

    // обычно не пишут
    return;
}
```

Вызов функции

Для вызова функции необходимо указать ее имя, за которым в круглых скобках через запятую перечислить аргументы.

```
a = avg(2.0, 5.0);

if (avg(a, b) < 0.0)
    printf("Averadge is negative!\n");

beep();
```

Если опустить скобки, функция не будет вызвана.

```
beep;   // warning: statement with no effect
```

Вызов функции

Значение, возвращаемое функцией, может быть проигнорировано.

```
int main(void)
{
    int n_chars;

    max(3, 5);    // странно, но допустимо

    n_chars = printf("Hello, world!\n");
    // после вызова printf n_chars равно 14
    printf("n_chars = %d\n", n_chars);

    (void) printf("Hello, world!\n");
    // явно указано, что возвращаемое значение не используется

    return 0;
}
```

Вызов функции

Операция	Название	Нотация	Класс	Приоритет	Ассоциат.
`( . . . )`	Вызов функции	`X(Y)`	Постфиксная	16	Слева направо

Объявление функции

```
#include <stdio.h>

int main(void)
{
    float a = avg(2.0, 3.0);
    // warning: implicit declaration of function 'avg`
    // error: conflicting types for 'avg'

    printf("%f\n", a);

    return 0;
}

float avg(float a, float b)
{
    return (a + b) / 2.0;
}
```

Объявление функции

Объявление функции предоставляет компилятору всю информацию, необходимую для вызова функции: количество и типы параметров, их последовательность, тип возвращаемого значения.

Объявление функции состоит из заголовка функции

```
тип-результата имя-функции(список формальных параметров
                                                с их типами);
```

Объявление функции должно соответствовать ее определению.

Объявление функции может не содержать имен параметров. Однако их обычно оставляют для большей наглядности.

Объявление функции

```
#include <stdio.h>

float avg(float a, float b);    // float avg(float, float);

int main(void)
{
    float a = avg(2.0, 3.0);

    printf("%f\n", a);

    return 0;
}

float avg(float a, float b)
{
    return (a + b) / 2.0;
}
```

Функции без параметров

```
#include <stdio.h>

void f()
{
    printf("f\n");
}

void g(void)
{
    printf("g\n");
}
```

```
int main(void)
{
    // ошибка компиляции?
    f();

    // ошибка компиляции?
    g();

    // ошибка компиляции?
    f(1, 2, 3);

    // ошибка компиляции?
    g(1, 2, 3);

    return 0;
}
```

Функции без параметров

Объявление

```
void f(void);
```

означает, что у функции нет ни одного параметра.

Объявление

```
void f();
```

означает, что у функции могут быть, и могут и не быть параметры. Если параметры есть, мы не знаем ни их количество, ни их тип.

Аргументы функции

Параметры функции указываются при ее определении в списке параметров. При этом в теле функции они представляют собой имена переменных, которые передаются в функцию при ее вызове.

Аргументы функции – это выражения, которые указываются при вызове функции.

```
int max(int a, int b)  // a и b – параметры функции max
{
    ...

c = max(n, m);         // n и m - аргументы
```

Аргументы функции

В Си все аргументы функции передаются «по значению».

Авторы языка: «Благодаря этому свойству обычно удается написать более компактную программу, содержащую меньшее число посторонних переменных, поскольку параметры можно рассматривать как должным образом инициализированные локальные переменные.»

Аргументы функции

```
#include <stdio.h>

int power(int base, int n)
{
    int result = 1;

    while (n-- > 0)
        result *= base;

    printf("n = %d\n", n);
    // n = -1

    return result;
}
```

```
int main(void)
{
    int a, n = 5;

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

    a = power(2, n);

    printf("%d^%d = %d\n",
                    2, n, a);
    // 2^5 = 32

    return 0;
}
```

Аргументы функции

Из-за такого способа передачи параметров возникают трудности при реализации функций, которые должны возвратить несколько параметров одновременно.

```
#include <stdio.h>

void decompose(
        float f,
          int int_part,
             float frac_part)
{
    int_part  = f;
    frac_part = f - int_part;
}
```

```
int main(void)
{
    int i;
    float f;

    decompose(3.14159, i, f);

    printf("%d %f\n", i, f);

    return 0;
}
```

Аргументы функции

Для решений этой проблемы необходимо каким-то образом предоставить функции decompose доступ к переменным, которые передаются ей в качестве аргументов.

Это можно сделать, передав в функцию не значения переменных, а адреса, по которым эти переменные располагаются в памяти.

Указатели (введение)

Переменная-указатель – это переменная, которая содержит адрес.

Переменная-указатель описывается как и обычная переменная. Единственное отличие – перед ее именем необходимо указать символ «*»:

```
int *p;
```

Это определение сообщает компилятору, что p – это переменная-указатель, которая может указывать на переменные целого типа.

Указатели (базовые операции)

В языке Си есть две операции, которые предназначены для использования именно с указателями.

Чтобы узнать адрес переменной, необходимо воспользоваться *операцией получения адреса* «&». Если x – это переменная, то &x – адрес переменной x в памяти.

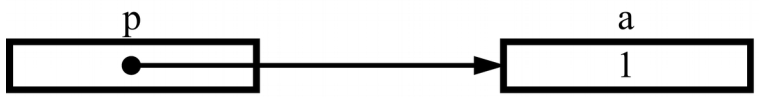
Чтобы получить доступ к объекту, на который указывает указатель, необходимо использовать *операцию разыменования* «*». Если p – переменная-указатель, то *p представляет объект, на который сейчас указывает p.

Указатели (базовые операции)

Операция	Название	Нотация	Класс	Приоритет	Ассоциат.
`&`	Адрес	`&X`	Префиксная	15	Справа налево
`*`	Разыменование	`*X`			

Пример использования базовых операций

```
int a = 1;
int *p = &a;                    // p теперь указывает на a
```




```
a = 3;

printf("%d %d\n", a, *p);       // 3 3

*p = 5;

printf("%d %d\n", a, *p);       // 5 5

printf("%p\n", p);              // адрес переменной a
```

Аргументы функции: использование указателей

```
#include <stdio.h>

void decompose(
        float f,
          int *int_part,
             float *frac_part)
{
  *int_part  = f;
  *frac_part = f - *int_part;
}
```

```
int main(void)
{
  int i;
  float f;

  decompose(3.14159, &i, &f);

  printf("%d %f\n", i, f);

  return 0;
}
```

Рекурсия

Функция называется *рекурсивной*, если она вызывает саму себя.

Например, следующая функция рекурсивно вычисляет факториал, используя формулу n! = n * (n - 1)!

```
int fact(int n)
{
    if (n == 0)
        return 1;

    return n * fact(n - 1);
}
```

Рекурсия

Отобразим последовательность запусков при вызове функции fact(3):

```
fact(3) = 3 * fact(2)      // приостановка выполнения fact(3)
  fact(2) = 2 * fact(1)    // приостановка выполнения fact(2)
    fact(1) = 1 * fact(0)  // приостановка выполнения fact(1)
      fact(0) = 1
    fact(1) = 1            // возобновление выполнения fact(1)
  fact(2) = 2              // возобновление выполнения fact(2)
fact(3) = 6                // возобновление выполнения fact(3)
```

Рекурсия

В рекурсии простейшей формы рекурсивный вызов расположен в конце функции. Такая рекурсия называется *хвостовой*.

Хвостовая рекурсия является простейшей формой рекурсии, поскольку она действует подобно циклу.

В большинстве случаев при реализации предпочтение отдается циклу:

- рекурсивный вызов использует больше памяти, поскольку создает свой набор переменных.
- рекурсия выполняется медленней, поскольку на каждый вызов функции требуется определенное время.