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

Дисциплина «Информатика» (основной уровень), 1-й курс, 1-й семестр.

Тематическое занятие 9

Функции: передача параметров.

Содержание

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 C 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;
}
```