



Тема 6

Подпрограммы

Понятие о подпрограммах

При решении задач довольно часто встречается ситуация, когда приходится многократно повторять одни и те же группы операторов, или требуется выполнить одни и те же вычисления, но с различными данными. Чтобы облегчить процесс программирования таких задач, в языки программирования было введено понятие подпрограмм.

Кроме того, подпрограммы удобно использовать для выделения логически завершённого участка кода, даже если он выполняется только один раз.

Определение подпрограммы

Подпрограмма – это именованная группа операторов, оформленная специальным образом, которая может вызываться по имени.

При **вызове** подпрограммы ей передаётся управление вычислительным процессом.

После выполнения подпрограммы осуществляется возврат на оператор вызывающей программы, следующий за вызовом подпрограммы. Такой процесс называется **выходом** из подпрограммы.

Процедуры и функции

Различают два вида подпрограмм:

- процедуры;
- функции.

Функции в момент выхода возвращают значение определённого типа (**результат** вызова), которое может быть использовано в вызывающей программе. Вызов функций, как правило, выполняется в выражениях.

Процедуры ничего не возвращают. Их вызов – отдельный оператор.

Функция может быть вызвана, как процедура (её результат теряется), но не наоборот.

Функции в C++

В языке C++ нет понятия процедур, есть только функции. Однако, если функция возвращает специальный тип **void**, она по сути является процедурой.

Функции могут быть **объявлены** и **определены**.

объявление_функции ::= заголовок;

определение_функции ::= заголовок {операторы ... }

заголовок ::= [модификаторы] тип_возврата имя_функции
([формальные_параметры])

формальный_параметр ::= [const] тип имя [= значение_по_умолчанию]

вызов_функции ::= имя_функции (фактические_параметры)

Формальные и фактические параметры

Формальные параметры, описанные в заголовке функции, играют роль дополнительных локальных переменных. Значения формальных параметров формируются при вызове функции, исходя из вычисленных значений фактических параметров.

При вызове функции фактический параметр преобразуется к типу формального.

Если формальный параметр описан с модификатором `const`, изменять его значения нельзя!

Если последние из фактических параметров не заданы, то соответствующие формальные параметры инициализируются значениями по умолчанию.

Пример определения и вызова функции

Задача: написать функцию, возвращающую максимальное из трёх чисел. Числа передаются в функцию через параметры.

```
int max3(int a, int b, int c) { //заголовок функции
    int max;
    if (a > b)
        max = a;
    else
        max = b;
    if (c > max)
        max = c;
    return max;
} //конец тела функции
```

Вызов этой функции в головной программе может выглядеть так:

```
int m, x=67, y=98, z=56;
cout << max3(x, y, z+50)<< endl; //106
cout << max3(9, 7, 111)<< endl; //111
```

Выход из функции

Выход из функции может быть выполнен двумя способами:

- выполнением специального оператора

return [выражение]

- после выполнения последнего оператора, входящего в тело функции (допускается только для функций, возвращающих void, и для функции main)

Допускается запись нескольких операторов return, задающих выход из разных ветвей вычислений.

Способы передачи параметров

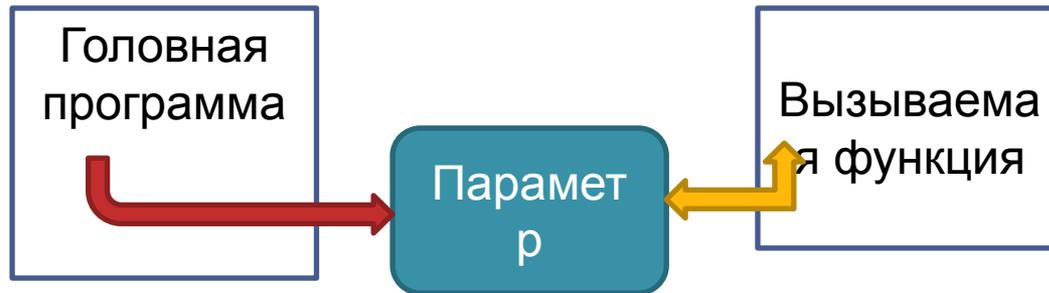
Различают две реализации передачи параметров в функцию:

- по значению;
- по адресу

При передаче параметра *по значению* перед вызовом функции выделяется специальный участок памяти, в которую заносится значение фактического параметра. При выходе из функции эта память освобождается. Изменения формальных параметров, переданных по значению, теряются при выходе!

При передаче параметров *по адресу* функция получает адрес тех данных, с которыми работает также и головная программа. Изменения сохраняются после выхода!

Схема организации передачи параметров



Передача параметров по значению

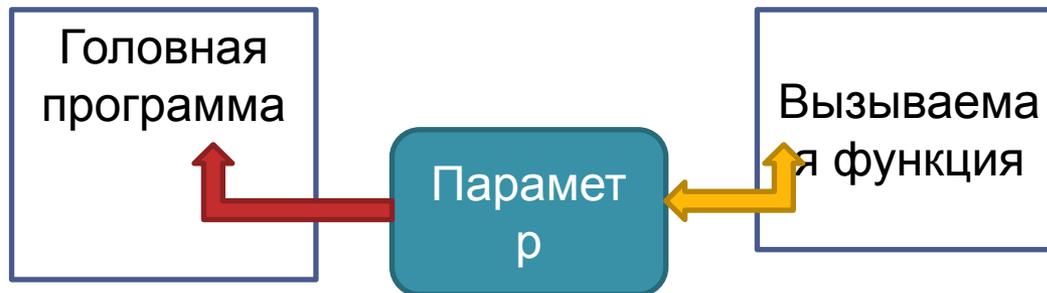


Передача параметров по адресу

Схема организации передачи параметров (продолжение)



Передача константных параметров по адресу



*Передача выходных параметров
(не поддерживается в C++)*

Особенности передачи параметров в С и С++

В языках С и С++ параметры передаются только по значению.

Преимущество: в качестве фактического параметра может быть записано любое выражение подходящего типа.

Недостаток: изменения формальных параметров теряются при выходе из вызываемой подпрограммы.

Пример неправильной работы с параметрами

Задача: написать функцию, меняющую местами значения своих параметров.

```
void Swap (int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

```
int x=10, y=100;
Swap(x, y);
// обмена нет!
```

Решение проблемы модификации параметров в С

Если нам необходимо изменить значение переданных в функцию данных, в качестве формального параметра должен быть задан указатель, а в качестве фактического – адресное выражение:

```
void Swap (int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

```
int x=10, y=100;
Swap (&x, &y);
// обмен выполнен!
```

Ссылочные типы C++

Ссылочный тип появился в языке C++ и используется главным образом при работе с модифицируемыми параметрами функций.

Ссылочный тип, как и указатель, основан на некотором базовом типе, его описание имеет вид:

`базовый_тип &`

Ссылку можно рассматривать как синоним имени, указанного при ее инициализации, или как указатель, который автоматически разыменовывается.

Ссылки в C++ должны быть связаны с каким-либо объектом. Таким образом, «нулевые ссылки» (и тип `void&`) в C++ отсутствуют.

Примеры работы со ссылочным ТИПОМ

```
int c = 100;  
int &p = c;  
// p и c - различные имена для одной переменной  
int &r; // ошибка - ссылка не может быть пустой  
p++; // переменная c также будет равна 101
```

Использование ссылочного типа для передачи параметров

Ссылочный тип может использоваться как в описании формальных параметров, так и при задании типа возвращаемой функции.

- Если формальный параметр – ссылка, то фактическим параметром должно быть Lvalue- выражение соответствующего типа. Модификации фактических параметров сохраняются после выхода из функции;
- Если функция возвращает ссылку, то результат её работы – Lvalue- выражение.

Пример использования ссылки в качестве параметра

Задача: написать функцию, меняющую местами значения своих параметров.

```
void Swap (int &a, int &b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

```
int x=10, y=100;
Swap(x, y);
// обмен выполнен!
```

Пример использования ссылки при выходе

```
int& my_inc(int &z) {  
    z++;  
    return z;  
}
```

...

```
int a = 1957;  
my_inc(a) = 20;  
cout << a << endl; // результат вывода - 20
```

Константные ссылки

Если требуется запретить изменение параметров во время выполнения функции и выдавать соответствующие сообщения на этапе компиляции, формальные параметры описываются с модификатором **const**. Ссылки также могут быть записаны с этим модификатором (т.н. **константные ссылки**)

Использование этого модификатора, во-первых, позволяет не строить громоздкие копии значений, а передавать лишь короткие адреса, и во-вторых не позволяет изменить значения фактических параметров.

Пример константной ссылки

```
void my_func(const long double &z) {  
    cout << z << endl;  
    return;  
}
```

```
void my_func(const long double z) {  
    cout << z << endl;  
    return;  
}
```

В первом случае в стеке выделяется 4 байта, во втором – 10!

Если формальный параметр – константная ссылка, то фактический уже не обязан быть Lvalue!

Побочный эффект подпрограмм

Подпрограммы имеют полный доступ к глобальным переменным, включая возможность их изменения. Изменение программных объектов, не переданных в качестве параметров, называется **побочным эффектом** подпрограмм.

Преимущества побочного эффекта:

- сокращается количество передаваемых параметров;
- заголовок функции остаётся неизменным

Недостатки побочного эффекта:

- можно случайно изменить глобальную переменную, предназначенную для совершенно других целей!

Использование побочного эффекта

```
void ReadData () {  
...  
// чтение данных и инициализация глобальных переменных  
}
```

Теперь ReadData() можно писать несколько раз, а изменение структуры данных и текста функции не влечёт изменение вызовов!

```
int k = 100;  
...  
void MyFunc () {  
...  
    for (k=0; k<20; k++) { ... }  
}
```

В качестве параметра цикла используется глобальная переменная, и её значение изменяется!

Рекурсия

Под рекурсией понимается вызов подпрограммы из тела этой же подпрограммы. Подобные соотношения достаточно часто встречаются в математике. Так, вычисление факториала можно представить следующим образом:

$$n! = \begin{cases} 1, & \text{если } n = 0 \\ n \cdot (n-1)!, & n > 0 \end{cases}$$

Для нахождения положительной степени числа можно использовать формулу

$$x^n = \begin{cases} 1, & \text{если } n = 0 \\ x \cdot x^{n-1}, & n > 0 \end{cases}$$

Такие соотношения называются **рекуррентными**.

Примеры рекурсивных подпрограмм

```
int Fact(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * Fact(n-1);  
}
```

```
double IntPower(double x; int n)  
{  
    if (n == 0)  
        return 1;  
    else  
        return x * IntPower(x, n-1);  
}
```

Примеры рекурсивных подпрограмм (продолжение)

Задача: ввести последовательность чисел, оканчивающуюся нулем и вывести их в обратном порядке.

```
void Reverse() {  
    int ch;  
    cin >> ch;  
    if (ch != 0)  
        Reverse();  
    cout << ch << "\n";  
}
```

Формы рекурсивных подпрограмм

1. Выполнение каких-то действий до рекурсивного вызова (на **рекурсивном спуске**)
2. Выполнение каких-то действий после рекурсивного вызова (на **рекурсивном возврате**)
3. Форма с выполнением действий как до, так и после рекурсивного вызова (с выполнением действий как на рекурсивном спуске, так и на рекурсивном возврате)

В общем случае любая рекурсивная функция включает в себя некоторое множество операторов S и один или несколько операторов рекурсивного вызова.

Глубина и текущий уровень рекурсии

- Максимальное число рекурсивных вызовов функции без возвратов, которое происходит во время выполнения программы, называется **глубиной рекурсии**.
- Число рекурсивных вызовов в каждый конкретный момент времени, называется **текущим уровнем рекурсии**.

Защипливание рекурсивных функций

- Рекурсивный вызов должен быть внутри какого-то условия, которое обязательно должно быть ложным!
- Защищипливание рекурсивных функций рано или поздно приведёт к ситуации «переполнение стека» (для каждой копии рекурсивной функции необходимо выделять дополнительную область памяти, а бесконечной памяти не существует).

Пример зацикливающейся рекурсивной функции

```
void Pech () {  
    cout << "У попа была собака\n";  
    cout << "Он её любил \n";  
    cout << "Она съела кусок мяса -\n";  
    cout << "Он её убил \n";  
    cout << "Закопал, закопал \n";  
    cout << "На могиле написал: \n";  
    Pech ();  
}
```

После компиляции выдаётся сообщение

warning C4717: 'Pech' : recursive on all control paths, function will cause runtime stack overflow

Максимальный уровень рекурсии - 4716

Задача о ханойских башнях

- Даны три стержня А, В, С. На стержне А находятся n дисков разного диаметра, пронумерованных сверху вниз. Каждый меньший диск находится на большем. Требуется переместить эти диски на стержень С, сохранив их взаиморасположение. Перемещать можно только по одному верхнему диску и нельзя класть больший диск на меньший.

Алгоритм решения задачи о ханойских башнях

Если $n=1$ то

1. Переместить этот единственный диск с A на C и остановиться.

иначе

2. Переместить верхние $n-1$ дисков с A на B , используя C как вспомогательный.

3. Переместить оставшийся нижний диск с A на C .

4. Переместить $n-1$ дисков с B на C , используя A как вспомогательный.

Неэффективность рекурсии

Если рекурсивная программа содержит несколько рекурсивных вызовов с разной глубиной, то она будет работать медленно (для одного и того же уровня вычисления будут выполнены несколько раз).

Пример

Известную в комбинаторике величину «число сочетаний из n элементов по k » можно вычислять как по рекурсивной, так и по нерекурсивной формуле:

$$C_n^k = \begin{cases} C_{n-1}^k + C_{n-1}^{k-1}, & \text{если } k \neq 1, k \neq n, \\ 1 & \text{в противном случае} \end{cases}$$

$$C_n^k = \frac{n!}{k!(n-k)!} = \frac{(n-t+1) \cdot (n-t+2) \cdot \dots \cdot n}{t \cdot (t-1) \cdot \dots \cdot 1}, \quad \text{где } t = \min(k, n-k)$$

Неэффективность рекурсии (продолжение)

- Первый вариант программы

```
int sochet1(int n, int k) {  
    int i, t, s;  
    if ((k==0) || (k==n))  
        return 1;  
    else {  
        if (n-k > k)  
            t = k;  
        else  
            t = n-k;  
        s = 1;  
        for (i = 1; i <= t; i++)  
            s = s*(n-i+1)/i;  
        return s;  
    }  
}
```

Неэффективность рекурсии (продолжение)

- Второй вариант программы

```
int sochet2(int n, int k) {  
    if ((k==0) || (k==n))  
        return 1;  
    else  
        return (sochet2(n-1, k) +  
                sochet2(n-1, k-1));  
}
```

Результаты вычислений для $n=30$, $k=20$

```
D:\cpp\aab\Debug>timer aab.exe
30045015
Time : 0.031
Memory : 324 Kb

D:\cpp\aab\Debug>timer aab.exe
30045015
Time : 7.531
Memory : 324 Kb

D:\cpp\aab\Debug>
```

1Левая 2Правая 3Имя 4Расшир 5Модиф 6Размер 7Несорт 8Создан 9Доступ 10Описа

Перегрузка функций

Часто удобно для функций, выполняющих один и тот же алгоритм для данных разных типов, иметь одинаковые имена.

Создание нескольких различных функций с одинаковым именем, но с различными спецификациями (количеством и/или типом аргументов, но не типом возвращаемого значения) называется **перегрузкой функций**.

При вызове функции компилятор сам подбирает наиболее подходящий, по его мнению, вариант.

Если точного соответствия не найдено, то выполняется преобразование типов.

Пример перегрузки функций

```
int Min(int a, int b ) {  
    return a < b ? a : b;  
}
```

Эта функция неприменима к типу `double`:

```
cout << Min(4.5, 6);
```

результат:

4

Добавляем перегруженные функции:

```
double Min(double a, double b ) {  
    return a < b ? a : b;  
}  
long long Min(long long a, long long b ) {  
    return a < b ? a : b;  
}
```

И Т.Д.

Пример перегрузки функций (продолжение)

Для отдельных типов реализация перегруженной функции может быть совершенно другой:

```
const char* Min(const char* a, const char* b) {  
    return strcmp(a, b) < 0 ? a : b;  
}
```

Пример перегрузки функций (окончание)

Если компилятор не может подобрать подходящую функцию, выдаётся ошибка компиляции:

```
int a = 1957;  
cout << Min(a+5, 42.8) << endl;
```

Сообщение:

error C2666: 'Min' : 3 overloads have similar conversions

d:\cpp\aab\prg.cpp(18): could be '__int64 Min(__int64,__int64)'

d:\cpp\aab\prg.cpp(15): or 'double Min(double,double)'

d:\cpp\aab\prg.cpp(11): or 'int Min(int,int)'

while trying to match the argument list '(int, double)'

Указатели на функции

Указатели на функции хранят адреса точек входа в функции. Операция разыменования, применённая к такому указателю, приводит к вызову функции.

Определение указателя:

`типовозврата (* имя) ([параметры]);`

Первые скобки нужны для изменения приоритета операций:

`типовозврата * имя ([параметры]);`

- объявление функции, возвращающей указатель

```
int (*pf) (int, int);
```

```
//определение указателя на функцию, возвращающую int
```

```
int *pf (int, int);
```

```
//объявление функции, возвращающей int *
```

Пример использования указателей на функции

```
int max(int a, int b) {  
    return (a>b ? a : b);  
}  
...  
int (*pf) (int, int);  
// это - указатель на функцию типа int с двумя  
// параметрами  
...  
pf = &max;  
pf = max;           // можно писать и так... ☺  
...  
int s=*pf(5,10);  
           // вызов функции max через указатель на нее  
int s=pf(5,10);  
           // можно писать и так... ☺
```

Callback - функции

Указатели на функции могут быть переданы в качестве параметров в другие функции, что позволяет выполнять разные вызовы в зависимости от условий выполнения.

Функции, указатели на которые передаются в другие функции, и никогда не вызываемые напрямую, называются callback-функциями.

Пример использования callback-функций

```
int max(int a, int b) {
    return (a>b ? a : b);
}
int min(int a, int b) {
    return (a<b ? a : b);
}
void PrintRezult (int f(int, int), int a, int b) {
    cout << "Result is: " << f(a, b) << endl;
}
...
int (*maxmin) (int, int);
...
if (...) //надо ВЫВОДИТЬ МИНИМУМ
    maxmin = min;
else
    maxmin = max;
...
PrintRezult(maxmin, 20, 40);
```

Функция main

Функция main – та функция, с запуска которой начинает работу консольное приложение, написанное на C++.

Форматы функции main:

```
void main(void);
```

```
int main(void);
```

```
void main(int argc, char *argv[]);
```

```
int main(int argc, char *argv[]);
```

```
// указаны традиционные имена для параметров
```

Первый и третий варианты не являются стандартными и не работают на некоторых платформах. Кроме того, писать void в скобках не обязательно. Таким образом, самое простое и легальное объявление функции main

```
int main();
```

Выход из функции main

Если в функции main отсутствует оператор return, то перед выходом из неё подразумевается

```
return 0;
```

Это – требования стандарта. Однако это реализовано не во всех системах. Например, в VC++ 6.0 приходится явно записывать этот оператор.

Значение, возвращаемое функцией main, воспринимается операционной системой, как **код завершения** программы. Часто считается, что нормально завершившаяся программа выдаёт код 0, а ненулевой код говорит о проблемах при её работе.

Передача параметров в функцию `main`

Наличие параметров позволяет передать в запускаемую программу данные *командной строки*: пользователь может при запуске программы указать несколько дополнительных параметров, разделяя их пробелами.

```
int main(int argc, char *argv[]);
```

Аргумент `argc` определяет, сколько параметров, включая имя программы, записано в командной строке. Массив нуль-терминированных строк `argv` хранит значение каждого параметра (`argv[0]` – имя ехе-файла с полным путем к нему, `argv[1]` – первый параметр и т.д.).

Пример передачи параметров в функцию main

Задача: в программу в качестве параметра передаётся имя файла для обработки. Если параметр не передан, программа должна запросить имя файла в диалоге.

```
int main(int argc, char *argv[]) {
    char FileName[40];
    setlocale(LC_ALL, ".1251");
    if (argc>2) {
        cout << "Слишком много параметров\n";
        return 1;
    }
    if (argc==1) {
        cout << "Введите имя обрабатываемого файла: \n";
        cin.getline(FileName, 40);
    }
    else
        strcpy(FileName, argv[1]);
    ...
}
```